INTRODUCTION AU LANGAGE C

```
char rahc
[ ]
=
"\n/"
,
redivider
[ ]
=
"Able was I ere I saw elbA"
,
*
deliver,reviled
=
1+1
,
niam ; main
( )
{/*\}
\*/
int tni
=
0x0
,
rahctup,putchar
( )
,LACEDx0 = 0xDECAL,
rof ; for
(;(int) (tni);)
(int) (tni)
= reviled ; deliver =
redivider
;
for ((int)(tni)++,++reviled;reviled* *deliver;deliver++,++(int)(tni)) rof
=
(int) -1- (tni)
;reviled--;--deliver;
(tni)  =  (int)
- 0xDECAL + LACEDx0 -
rof ; for
(reviled--,(int)--(tni);(int) (tni);(int)--(tni),--deliver)
rahctup = putchar
(reviled* *deliver)
;
rahctup * putchar
((char) * (rahc))
;
/*\
{\*/}
```

Bernard Cassagne

Bernard Cassagne

Introduction au langage C

norme ISO / ANSI

Laboratoire CLIPS
Université Joseph Fourier & CNRS
Grenoble

Un fichier PostScript contenant ce document est librement accessible par l'URL :
`ftp://ftp.imag.fr/pub/DOC.UNIX/C/Introduction_ANSI_C.ps`

Version de ce document : 2.1 de juin 1998

Table des matières

Avant-propos

Au sujet de l'auteur

Je suis ingénieur au CNRS et je travaille dans un laboratoire de recherche de l'université de Grenoble : le laboratoire CLIPS (`http://www-clips.imag.fr`). Toute notification d'erreur ou toute proposition d'amélioration de ce document sera la bienvenue à l'e-mail `Bernard.Cassagne@imag.fr`.

Au sujet de ce manuel

La littérature technique nous a habitué à deux styles d'écriture de manuels : le style « manuel de référence » et le style « guide de l'utilisateur » Les manuels de référence se donnent comme buts d'être exhaustifs et rigoureux. Les guides de l'utilisateur se donnent comme but d'être didactiques. Cette partition vient du fait qu'il est quasiment impossible sur des sujets complexes comme les langages de programmation d'être à la fois rigoureux et didactique. Pour s'en persuader il suffit de lire le texte d'une norme internationale.

Ce manuel se place dans la catégorie « guide de l'utilisateur » : son but est de permettre à une personne sachant programmer, d'acquérir les éléments fondamentaux du langage C. Ce manuel présente donc chaque notion selon une gradation des difficultés et ne cherche pas à être exhaustif. Il comporte de nombreux exemples, ainsi que des exercices dont la solution se trouve dans le corps du texte, mais commence toujours sur une page différente. Le lecteur peut donc au choix, ne lire les solutions qu'après avoir programmé sa solution personnelle, ou bien lire directement la solution comme si elle faisait partie du manuel.

Les notions supposées connues du lecteur

Les notions supposées connues du lecteur sont les concepts généraux concernant les langages de programmation. En fin du texte se trouve un glossaire qui regroupe des concepts généraux et certains concepts propres du langage C. En cas de rencontre d'un mot inconnu, le lecteur est invité à s'y reporter.

Un mot sur les problèmes de traduction

Le document de référence concernant le langage C est la norme ANSI définissant le langage. C'est un document écrit en anglais technique, ce qui pose des problèmes de traduction : comment traduire les néologismes inventés pour les besoins du langage ? De manière à ne pas dérouter les lecteurs français, je me suis imposé de respecter les choix de traduction d'un grand éditeur [1] de livres techniques, même quand je n'ai pas trouvé ces choix très heureux. J'ai donc utilisé *déclarateur*, *initialisateur* et *spécificateur* bien que me semble-t-il, ils râpent assez fort le palais quand on les prononce.

Conventions syntaxiques

Les règles de grammaires qui sont données dans le corps de ce manuel sont simplifiées (dans un but didactique) par rapport à la grammaire officielle du langage. Cependant, le lecteur trouvera à l'annexe D la grammaire sous une forme exhaustive et conforme à la norme ANSI. La typographie des règles suit les conventions suivantes :

1. les éléments terminaux du langage seront écrits dans une fonte à largeur constante, comme ceci : `while`.

2. les éléments non terminaux du langage seront écrits en italique, comme ceci : *instruction*.

3. les règles de grammaires seront écrites de la manière suivante :

   - les parties gauches de règles seront seules sur leur ligne, cadrées à gauche et suivies du signe deux points (:).
   - les différentes parties droites possibles seront introduites par le signe ⇒ et indentées sur la droite.
     Exemple :

     instruction :
     ⇒ `if` `(` *expression* `)` $instruction_1$
     ⇒ `if` `(` *expression* `)` $instruction_1$ `else` $instruction_2$

     Ceci signifie qu'il y a deux manières possibles de dériver le non-terminal *instruction*. La première règle indique qu'on peut le dériver en :
     `if` `(` *expression* `)` $instruction_1$
     la deuxième règle indique qu'on peut aussi le dériver en :
     `if` `(` *expression* `)` $instruction_1$ `else` $instruction_2$
   - une partie droite de règle pourra être écrite sur plusieurs lignes. Ceci permettra de refléter une manière possible de mettre en page le fragment de programme C correspondant, de façon à obtenir une bonne lisibilité.
     Sans en changer la signification, l'exemple précédent aurait pu être écrit :

     instruction :
     ⇒ `if` `(` *expression* `)`
     $instruction_1$
     ⇒ `if` `(` *expression* `)`
     $instruction_1$
     `else` $instruction_2$

1. C'est MASSON qui a édité en français [1] et [2] (Cf. Bibliographie)

- les éléments optionnels d'une règle seront indiqués en mettant le mot « option » (en italique et dans une fonte plus petite) à droite de l'élément concerné.
  Par exemple, la règle :

  déclarateur-init :
  $\Rightarrow$ *déclarateur* *initialisateur*$_{option}$

  indique que *déclarateur-init* peut se dériver soit en :

  déclarateur *initialisateur*

  soit en :

  déclarateur

Remerciements

Beaucoup de personnes m'ont aidé à améliorer le manuscrit original en me signalant de nombreuses erreurs et en me proposant des améliorations. Qu'elles soient toutes remerciées de leurs lectures attentives et amicalement critiques, tout particulièrement Damien Genthial, Fabienne Lagnier, Xavier Nicollin et Serge Rouveyrol.

Chapitre 1

Les bases

Le but de ce chapitre est de présenter les éléments de base du langage C, sous leur forme la plus simple. Arrivé au bout du chapitre, le lecteur sera capable d'écrire des programmes élémentaires.

1.1 Les versions du langage C

Le langage C a subi au cours de son histoire deux grandes étapes de définition. Il a été défini une première fois par deux chercheurs des Laboratoires Bell, B. Kernighan et D. Ritchie, dans un livre intitulé « The C Programming Language », publié en 1978. Cette version est appelée « Kernighan et Ritchie 78 », ou K&R 78 en abrégé, ou encore le plus souvent, simplement K&R.

Suite à l'extraordinaire succès d'UNIX, qui induisit le succès du langage C, la situation devint confuse : plusieurs fournisseurs de compilateurs mirent sur le marché des compilateurs non conformes à K&R car comportant des extensions particulières. À la fin des années 80, il devint nécessaire de mettre de l'ordre dans ce chaos et donc de normaliser le langage, tâche à laquelle s'attela l'ANSI[1], organisme de normalisation américain. La norme ANSI fut terminée en 1989. En 1990, l'ISO[2], organisme de normalisation international, (donc chapeautant l'ANSI), adopta tel quel le standard ANSI en tant que standard ISO.

Cette seconde version du langage C devrait donc s'appeler ISO C, mais comme les acteurs importants du monde informatique sont de culture anglo-saxonne et que ceux-ci persistent à l'appeler ANSI C, (presque ?) tout le monde fait de même. Dans ce manuel, nous suivrons l'usage général, et utiliserons l'expression ANSI C pour désigner la norme commune à l'ANSI et l'ISO.

Ce document décrit C ANSI, avec parfois des références à C K&R, de manière à permettre au lecteur de comprendre les sources écrits avant l'apparition de la norme.

1.2 Langage et bibliothèque standard

Le langage C a été conçu pour l'écriture de systèmes, en particulier le système UNIX. Pour cette raison, ses concepteurs ont fait une séparation nette entre ce qui est purement

1. American National Standards Institute
2. International Standards Organization

algorithmique (déclarations, instructions, etc.) et tout ce qui est interaction avec le système (entrées sorties, allocation de mémoire, etc.) qui est réalisé par appel de fonctions se trouvant dans une bibliothèque dite *bibliothèque standard*. Cette coupure se retrouve dans la norme qui est composée essentiellement de deux grands chapitres, les chapitres « langage » et « bibliothèque ».

Ce manuel se donne comme objectif de donner une vue d'ensemble du langage, mais pas de la bibliothèque standard. De la bibliothèque standard ne seront présentées de manière complète que les fonctions permettant de réaliser les entrées-sorties et la gestion mémoire. Cependant, la liste exhaustive des noms des fonctions de la bibliothèque, classés par type d'utilisation, est donnée dans le chapitre 10.

1.3 Les phases de compilation

Les compilateurs C font subir deux transformations aux programmes :

1. un *préprocesseur* réalise des transformations d'ordre purement textuel, pour rendre des services du type inclusion de source, compilation conditionnelle, et traitement de macros ;

2. le *compilateur* proprement dit prend le texte généré par le préprocesseur et le traduit en instructions machine.

La fonction de préprocesseur est assez souvent implémentée par un programme séparé (`cpp` sous UNIX) qui est automatiquement appelé par le compilateur.

1.4 Les jeux de caractères

Le lecteur non familiarisé avec les problèmes de codage de caractères peut se reporter à l'annexe A, où ces problèmes sont développés.

Le langage C n'impose pas un jeu de caractères particulier. Par contre tout le langage (mots-clés, opérateurs, etc.) est défini en utilisant les caractères ASCII. Même les identificateurs doivent être écrits avec l'alphabet anglais. Par contre, le jeu de caractères utilisé pour les constantes caractère, les chaînes de caractères et les commentaires est dépendant de l'implémentation.

Pendant très longtemps les programmeurs non anglophones ont utilisé l'ASCII faute de mieux, pour programmer en C. Actuellement, si on est dans le monde UNIX, il ne doit pas y avoir de problème pour disposer d'un environnement de travail (la fenêtre, le shell, l'éditeur, le compilateur) entièrement à la norme ISO-8859. Dans ce manuel, on suppose que le lecteur dispose d'un tel environnement : les exemples donnés sont écrits en ISO-8859.

1.5 Les unités lexicales

Le langage comprends 6 types d'unités lexicales : les mots-clés, les identificateurs, les constantes, les chaînes, les opérateurs et les signes de ponctuation.

1.5.1 Les mots-clés

Le langage C est un langage à mots-clés, ce qui signifie qu'un certain nombre de mots sont réservés pour le langage lui-même et ne peuvent donc pas être utilisés comme identificateurs. La liste exhaustive des mots-clés est la suivante :

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

Attention

Si le compilateur produit un message d'erreur syntaxique incompréhensible il est recommandé d'avoir le réflexe de consulter la liste des mots clés pour vérifier que l'on a pas pris comme identificateur un mot-clé. Si le lecteur désire être convaincu, il lui est suggéré de donner le nom `long` à une variable entière.

1.5.2 Les identificateurs

Le but d'un identificateur est de donner un nom à une entité du programme (variable, procédure, etc.) Les identificateurs sont formés d'une suite de lettres, de chiffres et du signe souligné, suite dont le premier caractère ne peut pas être un chiffre. Les lettres formant les identificateurs peuvent être majuscules ou minuscules, mais doivent faire partie de l'alphabet anglais : les lettres accentuées sont interdites. Les noms `var1`, `PremierIndex`, `i_tab`, `_deb` sont des identificateurs valides, mais `1i` et `i:j` ne le sont pas.

Un compilateur a le droit de tronquer les identificateurs internes (ceux qui ne sont pas exportés à un éditeur de liens) au delà d'une certaine longueur. Cette limite dépend de l'implémentation, mais ne doit pas être inférieure à 31 caractères.

De la même manière, les identificateurs externes (exportés à un éditeur de liens) pourront être tronqués au delà d'une certaine longueur. Cette limite est généralement plus sévère, mais ne peut être inférieure à 6 caractères. De surcroît, la distinction entre minuscules et majuscules n'est pas garantie (au contraire des noms internes, pour lesquels cette distinction est garantie).

1.6 Les commentaires

- Syntaxe :
  Les commentaires débutent par `/*` et se terminent par `*/`. Exemple :

  ```
  /*   Ceci est un commentaire   */
  ```

  Toute occurrence de `/*` est interprétée comme le début d'un commentaire sauf dans une chaîne littérale, ou un commentaire (les commentaires ne peuvent donc pas être imbriqués).

- Recommandations :
  Dans le domaine général de la programmation, (pas seulement le langage C), il est admis qu'il faille commenter selon les niveaux suivants :
  - unité de compilation : pour indiquer le nom de l'auteur, les droits de copyright, la date de création, les dates et auteurs des différentes modifications, ainsi que la raison d'être de l'unité ;
  - procédure : pour indiquer les paramètres et la raison d'être de la procédure ;
  - groupe d'intructions : pour exprimer ce que réalise une fraction significative d'une procédure ;
  - déclaration ou instruction : le plus bas niveau de commentaire.

  Pour le niveau unité de compilation, voici un exemple tiré du source de perl :

```
/*
 *    Copyright (c) 1991, Larry Wall
 *
 *    You may distribute under the terms of either the GNU General Public
 *    License or the Artistic License, as specified in the README file.
 *
 * $Log: perl.c,v $
 * Revision 4.0.1.8  1993/02/05  19:39:30  lwall
 * Revision 4.0.1.7  92/06/08  14:50:39  lwall
 * Revision 4.0.1.3  91/06/07  11:40:18  lwall
 * Revision 4.0.1.2  91/06/07  11:26:16  lwall
 * Revision 4.0.1.1  91/04/11  17:49:05  lwall
 * Revision 4.0  91/03/20  01:37:44  lwall
 * 4.0 baseline.
 *
 */
```

  Pour le niveau de la procédure, je trouve agréable de réaliser des espèces de cartouches permettant de découper visuellement un listing en ses différentes procédures, comme ceci par exemple :

```
/******************************************************************************/
/*                                                                            */
/*                               strcpy                                       */
/*                                                                            */
/*   But:                                                                     */
/*      copie une chaîne dans une autre                                       */
/*                                                                            */
/*   Interface:                                                               */
/*      s1 : chaîne destination                                               */
/*      s2 : chaîne source                                                    */
/*                                                                            */
/******************************************************************************/
```

  En ce qui concerne le niveau groupe d'instruction, il est classique de faire une mise en page comme dans l'exemple suivant tiré du source de dvips[3] :

```
/*
```

3. dvips est un traducteur en PostScript du format généré par TeX

```
 *   If nothing above worked, then we get desperate.  We attempt to
 *   open the stupid font at one of a small set of predefined sizes,
 *   and then use PostScript scaling to generate the correct size.
 *
 *   We much prefer scaling up to scaling down, since scaling down
 *   can omit character features, so we try the larger sizes first,
 *   and then work down.
 */
```

Pour le niveau déclaration ou instruction, on commentera sur la même ligne. Exemple tiré du source du compilateur GNU CC :

```
char *name;                   /* Function unit name.  */
struct function_unit *next;   /* Next function unit.  */
int multiplicity;             /* Number of units of this type.  */
int simultaneity;             /* Maximum number of simultaneous insns
                                 on this function unit or 0 if unlimited.  */
struct range ready_cost;      /* Range of ready cost values.  */
struct range issue_delay;     /* Range of issue delay values.  */
```

Attention

- L'erreur classique avec les commentaires est d'oublier la séquence fermante `*/`. Dans ce cas, le compilateur va considérer que le commentaire se poursuit jusqu'à la fin du prochain commentaire et ceci peut ne pas générer d'erreur syntaxique.

  Exemple :
  instruction
  `/*   premier commentaire`
  instruction
  ...
  instruction
  `/*   second commentaire   */`
  instruction

  On voit que dans ce cas, tout un ensemble d'instructions sera ignoré par le compilateur sans générer le moindre message d'erreur [4].

- Un commentaire ne peut pas contenir un commentaire, il n'est donc pas possible de mettre en commentaire un morceau de programme comportant déjà des commentaires [5].

4. Dans [5], Peter Van der Linden donne un exemple amusant de ce bug. Dans un compilateur, l'efficacité de l'algorithme de hachage de la table des identificateurs dépendait de la bonne initialisation d'une variable. Dans le code initial, l'initialisation avait été mise involontairement dans un commentaire provoquant une initialisation par défaut à zéro. La simple correction du bug, fit gagner 15% d'efficacité au compilateur !

5. Voir cependant F.4

1.7 Les types de base

1.7.1 Les caractères

Le mot-clé désignant les caractères est `char`. Un objet de ce type doit pouvoir contenir le code de n'importe quel caractère de l'ensemble des caractères utilisé sur la machine. Le codage des caractères n'est pas défini par le langage : c'est un choix d'implémentation. Cependant, dans la grande majorité des cas, le code utilisé est le code dit ASCII, ou un surensemble comme par exemple la norme ISO-8859.

Attention

Le type caractère est original par rapport à ce qui se fait habituellement dans les langages de programmation. La norme précise clairement qu'un objet de type caractère peut être utilisé dans toute expression où un objet de type entier peut être utilisé. Par exemple, si `c` est de type `char`, il est valide d'écrire `c + 1` : cela donnera le caractère suivant dans le code utilisé sur la machine.

1.7.2 Les entiers

Le mot clé désignant les entiers est `int`. Les entiers peuvent être affectés de deux types d'attributs : un attribut de précision et un attribut de représentation.

Les attributs de précision sont `short` et `long`. Du point de vue de la précision, on peut avoir trois types d'entiers : `short int`, `int` et `long int`. Les `int` sont implémentés sur ce qui est un mot « naturel » de la machine. Les `long int` sont implémentés si possible plus grands que les `int`, sinon comme des `int`. Les `short int` sont implémentés si possible plus courts que les `int`, sinon comme des `int`. Les implémentations classiques mettent les `short int` sur 16 bits, les `long int` sur 32 bits, et les `int` sur 16 ou 32 bits selon ce qui est le plus efficace.

L'attribut de représentation est `unsigned`. Du point de vue de la représentation, on peut avoir deux types d'entiers : `int` et `unsigned int`. Les `int` permettent de contenir des entiers signés, très généralement en représentation en complément à 2, bien que cela ne soit pas imposé par le langage. Les `unsigned int` permettent de contenir des entiers non signés en représentation binaire.

On peut combiner attribut de précision et attribut de représentation et avoir par exemple un `unsigned long int`. En résumé, on dispose donc de six types d'entiers : `int`, `short int`, `long int` (tous trois signés) et `unsigned int`, `unsigned short int` et `unsigned long int`.

1.7.3 Les flottants

Il existe trois types de flottants correspondants à trois précisions possibles. En allant de la précision la plus faible vers la plus forte, on dispose des types `float`, `double` et `long double`. La précision effectivement utilisée pour chacun de ces types dépend de l'implémentation.

1.8 Les constantes

1.8.1 Les constantes entières

- Syntaxe :
  On dispose de 3 notations pour les constantes entières : décimale, octale et hexadécimale.

  Les constantes décimales s'écrivent de la manière usuelle (ex : `372`). Les constantes octales doivent commencer par un zéro et ne comporter que des chiffres octaux (ex : `0477`). Les constantes hexadécimales doivent commencer par `0x` ou `0X` et être composées des chiffres de 0 à 9, ainsi que des lettres de a à f sous leur forme majuscule ou minuscule (ex : `0x5a2b`, `0X5a2b`, `0x5A2B`).

  Une constante entière peut être suffixée par la lettre `u` ou `U` pour indiquer qu'elle doit être interprétée comme étant non signée. Elle peut également être suffixée par la lettre `l` ou `L` pour lui donner l'attribut de précision `long`.

- Sémantique :
  Le type d'une constante entière est le premier type, choisi dans une liste de types, permettant de représenter la constante :

forme de la constante	liste de types
pas de suffixe, décimal	`int`, `long int`, `unsigned long int`
pas de suffixe, octal ou hexadécimal	`int`, `unsigned int`, `long int`, `unsigned long int`
suffixé par `u` ou `U`	`unsigned int`, `unsigned long int`
suffixé par `l` ou `L`	`long int`, `unsigned long int`
suffixé par (`u` ou `U`) et (`l` ou `L`)	`unsigned long int`

Attention

Ces conventions d'écriture des constantes ne respectent pas l'écriture mathématique, puisque `010` devant être interprété en octal, n'est pas égal à `10`.

1.8.2 Les constantes caractères

- Syntaxe :
  Une constante caractère s'écrit entourée du signe `'`. La règle générale consiste à écrire le caractère entouré du signe `'` ; par exemple, la constante caractère correspondant au caractère g s'écrit `'g'`.

Les cas particuliers

Les cas particuliers sont traités par une séquence d'échappement introduite par le caractère `\`.

- Caractères ne disposant pas de représentation imprimable.
  1. On peut les désigner par la notation `'\`*nb*`'` où *nb* est le code en octal du caractère. Exemple :

constante caractère	sémantique
`'\0'`	*null*
`'\12'`	*newline*
`'\15'`	*carriage return*
`'\33'`	*escape*

  2. On peut les désigner par la notation `'\x`*nb*`'` où *nb* est le code en hexadécimal du caractère. Exemple :

constante caractère	sémantique
`'\x0A'`	*newline*
`'\x0D'`	*return*
`'\x1B'`	*escape*

  3. Certains d'entre eux, utilisés très fréquemment, disposent d'une notation particulière. Il s'agit des caractères suivants :

constante caractère	sémantique
`'\n'`	*new line*
`'\t'`	*horizontal tabulation*
`'\v'`	*vertical tabulation*
`'\b'`	*back space*
`'\r'`	*carriage return*
`'\f'`	*form feed*
`'\a'`	*audible alert*

- Caractères disposant d'une représentation imprimable mais devant être désignés par une séquence d'échappement.

constante caractère	sémantique
`'\''`	'
`'\\'`	\

- Caractères disposant d'une représentation imprimable et pouvant être désignés soit par une séquence d'échappement soit par eux-mêmes.

constante caractère	sémantique
`'\"'` ou `'"'`	"
`'\?'` ou `'?'`	?

- Sémantique :
  Une constante caractère est de type `int` et a pour valeur le code du caractère dans le codage utilisé par la machine.

Note

Pourquoi diable les deux caractères `"` et `?` disposent ils de deux notations possibles?

- Le caractère `"` peut être représenté par la notation `'\"'` parce que celle-ci **doit** être utilisée dans les chaînes de caractères (voir plus loin 1.9). Pour des raisons de symétrie, les concepteurs du langage n'ont pas voulu qu'une notation valable pour une chaînes de caractères ne soit pas valable pour un caractère.

- Le caractère `?` est un cas à part à cause de l'existence des *trigraphes*. Les trigraphes sont des séquences de trois caractères permettant de désigner les caractères `# [ ] \ ^ { } | ~`. En effet, les terminaux conformes à la norme ISO 646:1983 ont remplacé ces caractères par des caractères nationaux. Les français, par exemple, connaissent bien le problème des `{` et des `}` qui se transforment en é et è.

  La norme ANSI a défini les 9 trigraphes suivants :

trigraphe	sémantique
`??=`	`#`
`??(`	`[`
`??)`	`]`
`??/`	`\`
`??'`	`^`
`??<`	`{`
`??>`	`}`
`??!`	`|`
`??-`	`~`

1.8.3 Les constantes flottantes

- Syntaxe :
  La notation utilisée est la notation classique par mantisse et exposant. La mantisse est composée d'une partie entière suivie du signe `.` (point) suivi de la partie fractionnaire. La partie entière et la partie fractionnaire sont exprimées en décimal et l'une ou l'autre peuvent être omises.

  L'exposant est introduit par la lettre e sous la forme minuscule ou majuscule. L'exposant est un nombre décimal éventuellement signé.

  Une constante flottante peut être suffixée par l'une quelconque des lettres `f`, `F`, `l`, `L`.

- Sémantique :
  Une constante non suffixée a le type `double`. Une constante suffixée par `f` ou `F` a le type `float`. Une constante suffixée par `l` ou `L` a le type `long double`.

  La valeur de la constante *mantisse* `e` *exposant* est *mantisse* $\times 10^{exposant}$.

  Si la valeur résultante ne correspond pas au type, la valeur est arrondie vers une valeur supérieure ou inférieure (le choix dépend de l'implémentation).

- Exemples :

notation C	notation mathématique
`2.`	2
`.3`	0.3
`2.3`	2.3
`2e4`	2×10^4
`2.e4`	2×10^4
`.3e4`	0.3×10^4
`2.3e4`	2.3×10^4
`2.3e-4`	2.3×10^{-4}

1.9 Les chaînes de caractères littérales

Une chaîne de caractères littérale est une suite de caractères entourés du signe `"`. Exemple :

```
"ceci est une chaîne"
```

Toutes les séquences d'échappement définies en 1.8.2 sont utilisables dans les chaînes. Exemple :

```
"ligne 1\nligne 2\nligne 3"
```

Le caractère `\` suivi d'un passage à la ligne suivante est ignoré. Cela permet de faire tenir les longues chaînes sur plusieurs lignes de source. Exemple :

```
"ceci est une très très longue chaîne que l'on fait tenir \
sur deux lignes de source"
```

Si deux chaînes littérales sont adjacentes dans le source, le compilateur concatène les deux chaînes. Exemple : `"Hello " "World!!"` est équivalent à `"Hello World!!"`.

Le compilateur rajoute à la fin de chaque chaîne un caractère mis à zéro. (Le caractère dont la valeur est zéro est appelé *null* dans le code ASCII). Cette convention de fin de chaîne est utilisée par les fonctions de la bibliothèque standard. Exemple : sur rencontre de `"Hello!"` le compilateur implantera en mémoire 7 caractères `H`, `e`, `l`, `l`, `o`, `!`, `\0`.

Dans une chaîne de caractères littérale, le caractère `"` *doit* être désigné par la séquence d'échappement, alors que `'` *peut* être désigné par sa séquence d'échappement ou par lui-même.

1.10 Les constantes nommées

Il y a deux façons de donner un nom à une constante : soit en utilisant les possibilités du préprocesseur, soit en utilisant des énumérations.

1.10.1 Les #define

Lorsque le préprocesseur lit une ligne du type :

`#define` *identificateur* *reste-de-la-ligne*

il remplace dans toute la suite du source, toute nouvelle occurrence de *identificateur* par *reste-de-la-ligne*. Par exemple on peut écrire :

```
#define PI 3.14159
```

et dans la suite du programme on pourra utiliser le nom `PI` pour désigner la constante `3.14159`.

Attention

Une telle définition de constante n'est pas une déclaration mais une commande du préprocesseur. Il n'y a donc pas de ; à la fin. Rappelons que le préprocesseur ne compile pas, il fait des transformations d'ordre purement textuel. Si on écrit :

```
#define PI 3.14159;
```

le préprocesseur remplacera toute utilisation de `PI` par `3.14159;` et par exemple, remplacera l'expression `PI / 2` par `3.14159; / 2` ce qui est une expression incorrecte. Dans une telle situation, le message d'erreur ne sera pas émis sur la ligne fautive (le `#define`), mais sur une ligne correcte (celle qui contient l'expression `PI / 2`), ce qui gênera la détection de l'erreur.

1.10.2 Les énumérations

On peut définir des constantes de la manière suivante :
`enum {` *liste-d'identificateurs* `}`
Par exemple :

```
enum {LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE};
```

définit les identificateurs `LUNDI`, ... `DIMANCHE` comme étant des constantes de type `int`, et leur donne les valeurs 0, 1, ... 6. Si on désire donner des valeurs particulières aux constantes, cela est possible :

```
enum {FRANCE = 10, ESPAGNE = 20, ITALIE = 30};
```

Il n'est pas nécessaire de donner une valeur à toutes les constantes :

```
enum {FRANCE = 10, LUXEMBOURG, BELGIQUE, ESPAGNE = 20, ITALIE = 30};
```

donnera la valeur 11 à `LUXEMBOURG` et 12 à `BELGIQUE`.

Remarque

Il est d'usage (au moins dans le monde UNIX) de donner un nom entièrement en majuscules aux constantes nommées d'un programme. Mon opinion est que ceci est une bonne convention, qui accroît la lisibilité des programmes.

1.11 Déclarations de variables ayant un type de base

Une telle déclaration se fait en faisant suivre un type par une liste de noms de variables. Exemple :

```
int i;          /*  déclaration de la variable i de type int               */
int i,j;        /*  déclaration de deux variables i et j de type int       */
short int k;    /*  déclaration de la variable k de type short int         */
float f;        /*  déclaration de la variable f de type float             */
double d1,d2;   /*  déclaration de deux variables d1 et d2 de type double  */
```

Il est possible de donner une valeur initiale aux variables ainsi déclarées. Exemple :

```
int i = 54;
int i = 34, j = 12;
```

1.12 Les opérateurs les plus usuels

1.12.1 L'affectation

En C, l'affectation est un opérateur et non pas une instruction.

- Syntaxe :

  expression :
  ⇒ *lvalue* `=` *expression*

  Dans le jargon C, une *lvalue* est une expression qui doit délivrer une variable (par opposition à une constante). Une *lvalue* peut être par exemple une variable simple, un élément de tableau, mais pas une constante. Cette notion permet d'exprimer dans la grammaire l'impossibilité d'écrire des choses du genre `1 = i` qui n'ont pas de sens.

  Exemples d'affectation :

  ```
  i = 3
  f = 3.4
  i = j + 1
  ```

- Sémantique :

  L'opérateur d'affectation a deux effets :

  1. il réalise un effet de bord consistant à affecter la valeur de *expression* à la variable désignée par la *lvalue* ;
  2. il délivre la valeur ainsi affectée, valeur qui pourra être utilisée dans une expression englobant l'affectation.

  Exemple :

  ```
  i = (j = k) + 1
  ```

La valeur de `k` est affectée à `j` et cette valeur est le résultat de l'expression `(j = k)` ; on y ajoute `1` et le résultat est affecté à `i`.

- Conversions de type :

  Lorsque la valeur de l'expression est affectée à la *lvalue*, la valeur est éventuellement convertie dans le type de la *lvalue*. On peut par exemple affecter une expression entière à un flottant.

1.12.2 L'addition

- Syntaxe :

  expression :
  $\Rightarrow$ `+` *expression*
  $\Rightarrow$ $expression_1$ `+` $expression_2$

- Sémantique :

  Les deux expressions sont évaluées, l'addition réalisée, et la valeur obtenue est la valeur de l'expression d'addition. (La sémantique de `+` *expression* est celle de `0 +` *expression*).

  L'ordre dans lequel les deux expressions sont évaluées, n'est pas déterminé. Si $expression_1$ et $expression_2$ font des effets de bords, on n'est donc pas assuré de l'ordre dans lequel ils se feront.

  Après évaluation des expressions, il peut y avoir conversion de type de l'un des opérandes, de manière à permettre l'addition. On pourra par exemple faire la somme d'une expression délivrant un flottant et d'une expression délivrant un entier : l'entier sera converti en flottant et l'addition sera réalisée entre flottants.

1.12.3 La soustraction

- Syntaxe :

  L'opérateur peut être utilisé de manière unaire ou binaire :

  expression :
  $\Rightarrow$ `-` *expression*
  $\Rightarrow$ $expression_1$ `-` $expression_2$

- Sémantique :

  Les deux expressions sont évaluées, la soustraction réalisée, et la valeur obtenue est la valeur de l'expression soustraction. (La sémantique de `-` *expression* est celle de `0 -` *expression*).

  Les mêmes remarques concernant l'ordre d'évaluation des opérandes ainsi que les éventuelles conversions de type faites au sujet de l'addition s'appliquent à la soustraction.

1.12.4 La multiplication

- Syntaxe :

  expression :
  $\Rightarrow$ $expression_1$ `*` $expression_2$

- Sémantique :

  Les deux expressions sont évaluées, la multiplication réalisée, et la valeur obtenue est la valeur de l'expression multiplicative.

  Les mêmes remarques concernant l'ordre d'évaluation des opérandes ainsi que les éventuelles conversions de type faites au sujet de l'addition s'appliquent à la multiplication.

1.12.5 La division

- Syntaxe :

  expression :
  $\Rightarrow$ $expression_1$ `/` $expression_2$

- Sémantique :

  Contrairement à d'autres langages, le langage C ne dispose que d'une seule notation pour désigner deux opérateurs différents : le signe `/` désigne à la fois la division entière et la division entre flottants.

  Si $expression_1$ et $expression_2$ délivrent deux valeurs entières, alors il s'agit d'une division entière. Si l'une des deux expressions au moins délivre une valeur flottante, il s'agit d'une division entre flottants.

  Dans le cas de la division entière, si les deux opérandes sont positifs, l'arrondi se fait vers zéro, mais si au moins un des deux opérandes est négatif, la façon dont se fait l'arrondi dépend de l'implémentation (mais il est généralement fait vers zéro). Exemple : `13 / 2` délivre la valeur 6 et `-13 / 2` ou `13 / -2` peuvent délivrer -6 ou -7 mais le résultat sera généralement -6.

  Les remarques concernant l'ordre d'évaluation des opérandes faites au sujet de l'addition s'appliquent également à la division. Les remarques concernant les éventuelles conversion de type faites au sujet de l'addition s'appliquent à la division entre flottants.

1.12.6 L'opérateur modulo

- Syntaxe :

  expression :
  $\Rightarrow$ $expression_1$ `%` $expression_2$

- Sémantique :

  Les deux expressions sont évaluées et doivent délivrer une valeur de type entier, on évalue le reste de la division entière de $expression_1$ par $expression_2$ et la valeur obtenue est la valeur de l'expression modulo.

Si au moins un des deux opérandes est négatif, le signe du reste dépend de l'implémentation, mais il est généralement pris du même signe que le dividende. Exemples :
`13 % 2` délivre 1
`-13 % 2` délivre généralement -1
`13 % -2` délivre généralement 1.

Les choix d'implémentation faits pour les opérateurs division entière et modulo doivent être cohérents, en ce sens que l'expression :
`b * (a / b) + a % b` (où `a` et `b` sont des entiers)
doit avoir pour valeur `a`.

1.12.7 Les opérateurs de comparaison

- Syntaxe :

  expression :
  ⇒ $expression_1$ *opérateur* $expression_2$
  où *opérateur* peut être l'un des symboles suivants :

opérateur	sémantique
`>`	strictement supérieur
`<`	strictement inférieur
`>=`	supérieur ou égal
`<=`	inférieur ou égal
`==`	égal
`!=`	différent

- Sémantique :

  Les deux expressions sont évaluées puis comparées, la valeur rendue est de type `int` et vaut 1 si la condition est vraie, et 0 sinon.

  On remarquera que le type du résultat est `int`, car le type booléen n'existe pas.

1.13 Les instructions les plus usuelles

1.13.1 Instruction expression

- Syntaxe :

  instruction :
  ⇒ *expression* `;`

- Sémantique :

  L'expression est évaluée, et sa valeur est ignorée. Ceci n'a donc de sens que si l'expression réalise un effet de bord. Dans la majorité des cas, il s'agira d'une expression d'affectation. Exemple :

```
i = j + 1;
```

Remarque

D'après la syntaxe, on voit qu'il est parfaitement valide d'écrire l'instruction
`i + 1;`
mais ceci ne faisant aucun effet de bord, cette instruction n'a aucune utilité.

1.13.2 Instruction composée

- Syntaxe :

  instruction :
  $\Rightarrow$ `{`
  liste-de-déclarations$_{option}$
  liste-d'instructions
  `}`

- Sémantique :

  Le but de l'instruction composée est double, elle permet :

  1. de grouper un ensemble d'instructions en lui donnant la forme syntaxique d'une seule instruction ;
  2. de déclarer des variables qui ne seront accessible qu'à l'intérieur de l'instruction composée (structure classique de 'blocs').

Remarques sur la syntaxe

- il n'y a pas de séparateur dans *liste-d'instructions* (les points virgules sont des terminateurs pour les instructions qui sont des *expressions*) ;
- les accolades jouent le rôle des mots clés `begin` et `end` que l'on trouve dans certains langages (Pascal, PL/1, etc.).

1.13.3 Instruction `if`

- Syntaxe :

  instruction :
  $\Rightarrow$ `if` `(` *expression* `)` *instruction*$_1$
  $\Rightarrow$ `if` `(` *expression* `)` *instruction*$_1$ `else` *instruction*$_2$

- Sémantique :

  expression est évaluée, si la valeur rendue est non nulle, on exécute *instruction*$_1$, sinon on exécute *instruction*$_2$ si elle existe.

Remarques sur la syntaxe

1. Attention au fait que *expression* doit être parenthésée ;
2. La partie then de l'instruction n'est pas introduite par un mot clé : pas de `then` comme dans certains langages.

3. Lorsqu'il y a ambiguïté sur l'instruction `if` dont dépend une partie `else`, l'ambiguïté est levée en faisant dépendre le `else` de l'instruction `if` la plus proche.

   Par exemple, si on écrit :

   ```
   if (a > b) if (c < d) u = v; else i = j;
   ```

   le `else` sera celui du `if (c <d)`. Si on voulait qu'il en soit autrement, il faudrait écrire :

   ```
   if (a > b)
      {
      if (c < d) u = v;
      }
   else i = j;
   ```

Remarques sur la sémantique

Étant donné que l'instruction `if` teste l'égalité à zéro de *expression*, celle-ci n'est pas nécessairement une expression de comparaison. Toute expression délivrant une valeur pouvant être comparée à zéro est valide.

Exemples d'instructions if

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

if (x > y)
   {
   ...          /*   liste d'instructions       */
   }
else
   {
   ...          /*   liste d'instructions       */
   }

if (a)          /*   équivalent à if (a != 0)   */
   {
   ...
   }
```

1.14 Inclusion de source

Nous avons déjà vu que la première phase de compilation est le traitement réalisé par le *préprocesseur* qui rend des services d'inclusion de source, de compilation conditionnelle et de traitement de macros. Nous allons voir dans ce chapitre ses capacités d'inclusion de source.

Lorsqu'on développe un gros logiciel, il est généralement nécessaire de le découper en unités de compilation de taille raisonnable. Une fois un tel découpage réalisé, il est courant que plusieurs unités aient certaines parties communes (par exemple des définitions de constantes à l'aide de commandes `#define`).

De façon à éviter la répétition de ces parties communes, le langage C offre une facilité d'inclusion de source qui est réalisée à l'aide de la commande `#include` du préprocesseur. Lorsque le préprocesseur rencontre une ligne du type :

`#include "` *nom-de-fichier* `"`

ou

`#include <` *nom-de-fichier* `>`

il remplace cette ligne par le contenu du fichier *nom-de-fichier*.

La manière dont se fait la recherche du fichier à inclure est dépendante de l'implémentation. Dans le système UNIX, il est traditionnel de se conformer à la règle suivante :

- si on utilise `<` et `>`, le fichier est un fichier système, il sera recherché dans un ou plusieurs répertoires systèmes connus du compilateur, comme par exemple `/usr/include`.
- si on utilise `"`, le fichier à inclure est un fichier utilisateur, il sera recherché dans l'espace de fichiers de l'utilisateur, par exemple le répertoire courant.

1.15 Les procédures et les fonctions

1.15.1 Définition d'une fonction

- Syntaxe :

  définition-de-fonction :

  ⇒ *type* *identificateur* `(` *liste-de-déclarations-de-paramètres* `)`
  `{`
  liste-de-déclarations$_{option}$
  liste-d'instructions
  `}`

  Note

  Dans le jargon C, l'ensemble :
  type *identificateur* `(` *liste-de-déclarations-de-paramètres* `)`
  porte le nom bizarre de *prototype de fonction*.

- Sémantique :

  type est le type de la valeur rendue par la fonction ; *identificateur* est le nom de la fonction ; *liste-de-déclarations-de-paramètres* est la liste (séparés par des virgules) des déclarations des paramètres formels. La *liste-de-déclarations*$_{option}$ permet si besoin est, de déclarer des variables qui seront locales à la fonction, elles seront donc inaccessibles de l'extérieur. La *liste-d'instructions* est l'ensemble des instructions qui seront exécutées sur appel de la fonction. Parmi ces instructions, il doit y avoir au moins une instruction du type :

  `return` *expression* `;`

  Lors de l'exécution d'une telle instruction, *expression* est évaluée, et le contrôle d'exécution est rendu à l'appelant de la fonction. La valeur rendue par la fonction est celle de *expression*.

Exemple

```
int sum_square(int i,int j)  /*   la fonction sum_square délivre un int      */
                             /*  ses paramètres formels sont les int i et j  */
{
int resultat;                /*   déclaration des variables locales          */

resultat = i*i + j*j;
return(resultat);            /*  retour a l'appelant en délivrant résultat   */
}
```

Instruction `return`

L'instruction `return` est une instruction comme une autre, il est donc possible d'en utiliser autant qu'on le désire dans le corps d'une fonction. Exemple :

```
int max(int i,int j)       /*   la fonction max délivre un int                */
                           /*   ses paramètres formels sont les int i et j    */
{                          /*   pas de variables locales pour max             */

if (i > j) return(i); else return(j);
}
```

Si la dernière instruction exécutée par une fonction n'est pas une instruction `return`, la valeur rendue par la fonction est indéterminée.

Les paramètres formels

Dans le cas où une fonction n'a pas de paramètres formels, le mot clé `void` est mis en tant que *liste-de-déclarations-de-paramètres*. Exemple :

```
double pi(void)              /*   pas de paramètres formels    */
{                            /*   pas de variables locales     */
return(3.14159);
}
```

Attention

La syntaxe des déclarations pour les paramètres formels et les variables n'est pas la même. Quand on a besoin de déclarer deux variables du même type, on peut utiliser deux déclarations :

```
int i;
int j;
```

Mais on peut aussi n'utiliser qu'une déclaration pour déclarer les deux variables :

```
int i,j;
```

Cette possibilité n'existe pas pour une liste de paramètres formels. Il n'est pas possible d'écrire `max` de la manière suivante :

```
int max(int i,j)   /*   incorrect   */
{
...
}
```

1.15.2 Appel d'une fonction

- Syntaxe :

  expression :

  ⇒ *identificateur* (*liste-d'expressions*)

- Sémantique :

  Les expressions de *liste-d'expressions* sont évaluées, puis passées en tant que paramètres effectifs à la fonction de nom *identificateur*, qui est ensuite exécutée. La valeur rendue par la fonction est la valeur de l'expression appel de fonction.

- Exemple :

  ```
  {
  int a,b,m,s;
  double d;

  s = sum_square(a,b);          /*   appel de sum_square   */
  m = max(a,b);                 /*   appel de max          */
  d = pi();                     /*   appel de pi           */
  }
  ```

Attention

1. Dans le cas d'une fonction sans paramètre, la *liste-d'expressions* doit être vide : il n'est pas possible d'utiliser le mot clé `void` en tant que paramètre effectif.

   ```
   d = pi(void);                 /*   appel incorrect de pi   */
   ```

2. L'ordre d'évaluation des paramètres effectifs n'est pas spécifié. Si certains de ceux-ci réalisent des effets de bords, l'ordre dans lequel ils se feront n'est pas garanti. Exemple :

   ```
   sum_square(f(x),g(y));
   ```

   La fonction `g` sera peut-être exécutée avant `f`.

1.15.3 Les procédures

Le langage C ne comporte pas à strictement parler le concept de procédure. Cependant, les fonctions pouvant réaliser sans aucune restriction tout effet de bord qu'elles désirent, le programmeur peut réaliser une procédure à l'aide d'une fonction qui ne rendra aucune valeur. Pour exprimer l'idée de « aucune valeur », on utilise le mot-clé `void`. Une procédure sera donc implémentée sous la forme d'une fonction retournant `void` et dont la partie *liste-d'instructions* ne comportera pas d'instruction `return`.

Lors de l'appel de la procédure, il faudra ignorer la valeur rendue c'est à dire ne pas l'englober dans une expression.

Exemple

```
void print_add(int i,int j)  /*  la procédure et ses paramètres formels   */
{
int r;                       /*  une variable locale à print_add          */

r = i + j;
...                          /*  instruction pour imprimer la valeur de r */
}

void prints(void)            /*   une procédure sans paramètres           */
{
int a,b;                     /*   variables locales à prints              */

a = 12; b = 45;
print_add(a,b);              /*   appel de print_add                      */
print_add(13,67);            /*   un autre appel à print_add              */
}
```

Problème de vocabulaire

Dans la suite du texte, nous utiliserons le terme de *fonction* pour désigner indifféremment une procédure ou une fonction, chaque fois qu'il ne sera pas nécessaire de faire la distinction entre les deux.

1.15.4 Fonctions imbriquées

À l'inverse de certains langages, les fonctions imbriquées n'existent pas dans le langage C. Il n'est donc pas possible qu'une fonction ne soit connue qu'à l'intérieur d'une autre fonction.

1.15.5 Récursivité

Il n'y a rien de spécial à faire à la déclaration d'une fonction pour qu'elle puisse être appelée de manière récursive. Voici un exemple de factorielle :

```
int facto(int n)
{
```

```
if (n == 1) return(1);
else return(n * facto(n-1));
}
```

1.15.6 Référence à une fonction externe

Quand on désire utiliser une fonction qui est définie ailleurs, il est nécessaire de la déclarer comme étant externe. Cela se fait en préfixant l'en-tête de la fonction du mot clé `extern`, comme ceci :

```
extern int sum_square(int i, int j);
```

Le cas le plus courant d'utilisation de fonction définie ailleurs est l'utilisation des fonctions de la bibliothèque standard. Avant d'utiliser une fonction de la bibliothèque standard, il faudra donc la déclarer en fonction externe. Il y a une méthode permettant de faire cela de manière automatique grâce au mécanisme d'inclusion de source du préprocesseur. Cela est expliqué dans la documentation de chaque fonction.

1.15.7 Comprendre la documentation de la bibliothèque standard

Dans la documentation concernant les fonctions de la bibliothèque standard, que ce soit dans la norme ou dans les manuels UNIX, l'information concernant l'interface d'appel de la fonction se trouve dans le paragraphe **Synopsis**. Dans ces paragraphes, se trouve toujours une commande d'inclusion de source. Exemple de la fonction *cosinus* :
Synopsis

```
#include <math.h>
double cos(double x);
```

Il faut comprendre que le fichier `math.h` contient un ensemble de définitions nécessaires à l'utilisation de `cos`, en particulier la déclaration de `cos` en fonction externe, à savoir :

```
extern double cos(double x);
```

Il faut donc inclure le fichier `math.h` avant toute utilisation de la fonction `cos`.

1.15.8 Les fonctions dans le style K&R

La définition des fonctions est la plus importante différence entre ANSI et K&R. Dans la version K&R du langage, le prototype (c'est à dire la *liste-de-déclarations-de-paramètres*) est remplacé par une liste d'identificateurs, qui sont les noms des paramètres. La déclaration des types des paramètres se fait juste après, de la manière suivante :

```
int sum_square(i,j)        /*  liste des noms des paramètres formels     */
int i, int j;              /*  déclaration du type des paramètres        */
{
int resultat;

resultat = i*i + j*j;
return(resultat);
}
```

En ce qui concerne la déclaration de fonction externe, il faut préfixer avec le mot-clé `extern`, l'en-tête de la fonction débarrassée de la liste des noms des paramètres, comme ceci :

```
extern int sum_square();
```

Dans ce cas, le compilateur ne connaît pas le type des paramètres de la fonction, il lui est donc impossible de détecter une erreur portant sur le type des paramètres lors d'un appel de fonction externe. Cette situation est assez catastrophique et c'est la raison pour laquelle le comité de normalisation a introduit le concept de prototype de fonction vu au chapitre 1.15.1. Il n'a cependant pas voulu faire un changement incompatible, il a donc décidé que les **deux méthodes** étaient acceptées, en précisant toutefois que la méthode K&R était une *caractéristique obsolescente*. En clair, cela signifie que cette méthode sera abandonnée dans la prochaine révision de la norme, si prochaine révision il y a.

Remarque

Nous ne pouvons que conseiller l'utilisation exclusive des prototypes de fonctions, et de ne jamais utiliser la méthode K&R. Si nous avons exposé celle-ci, c'est pour permettre au lecteur de comprendre les quantités énormes de source C existant, qui sont écrites à la K&R.

1.16 Impression formattée

Pour faire des entrées-sorties, il est nécessaire de faire appel aux possibilités de la bibliothèque standard. Celle-ci comporte une procédure permettant de réaliser des sorties formattées : il s'agit de `printf`. On l'appelle de la manière suivante :
`printf ( ` *chaîne-de-caractères* `, ` *liste-d'expressions* ` ) ;`
chaîne-de-caractères est le texte à imprimer dans lequel on peut librement mettre des séquences d'échappement qui indiquent le format selon lequel on veut imprimer la valeur des expressions se trouvant dans *liste-d'expressions*. Ces séquences d'échappement sont composée du caractère `%` suivi d'un caractère qui indique le format d'impression. Il existe entre autres, `%c` pour un caractère, `%d` pour un entier à imprimer en décimal, `%x` pour un entier à imprimer en hexadécimal. Exemple :

```
int i = 12;
int j = 32;
printf("la valeur de i est %d et celle de j est %d",i,j);
```

imprimera :

```
la valeur de i est 12 et celle de j est 32
```

Il est possible d'avoir une *liste-d'expressions* vide, dans ce cas l'appel à `printf` devient :
`printf ( ` *chaîne-de-caractères* ` ) ;`
par exemple, pour imprimer le mot erreur en le soulignant[6], on peut écrire :

```
printf("erreur\b\b\b\b\b\b______");   /*   \b est back-space   */
```

6. à condition que la sortie se fasse sur un terminal papier

Fichier d'include

La déclaration de `printf` en tant que fonction externe se trouve dans le fichier `stdio.h` qu'il faudra inclure avant toute utilisation. En tête du programme il faudra donc mettre :

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

1.17 Structure d'un programme

Nous ne considérerons pour débuter que le cas de programmes formés d'une seule unité de compilation. Un programme C est une suite de déclarations de variables et de définitions de fonctions dans un ordre quelconque. Les variables sont des variables dont la durée de vie est égale à celle du programme, et qui sont accessibles par toutes les fonctions. Ces variables sont dites variables globales, par opposition aux variables déclarées à l'intérieur des fonctions qui sont dites locales.

Bien que ce ne soit pas obligatoire, il est généralement considéré comme étant un bon style de programmation de regrouper toutes les déclarations de variables en tête du programme. La structure d'un programme devient alors la suivante :

programme :
$\Rightarrow$ *liste-de-déclarations*$_{option}$
liste-de-définitions-de-fonctions

Il peut n'y avoir aucune déclaration de variable, mais il doit y avoir au moins la définition d'une procédure dont le nom soit `main`, car pour lancer un programme C, le système appelle la procédure de nom `main`. Exemple :

```
int i,j;                    /*   i,j,a,b,c   sont des variables globales    */
double a,b,c;

void p1(int k)              /*   début de la définition de la procédure p1  */
                            /*   p1 a un seul paramètre entier : k          */
{
int s,t;                    /*   variables locales à p1                     */

...                         /*   instructions de p1 qui peuvent accéder à   */
...                         /*   k,i,j,a,b,c,s et t                         */
}                           /*   fin de la définition de p1                 */

int f1(double x,double y)   /*   début de la définition de la fonction f1   */
                            /*   f1 a deux paramètres flottants: x et y     */
{
double u,v;                 /*   variables locales à f1                     */

...                         /*   instructions de f1 qui peuvent accéder à   */
...                         /*   x,y,i,j,a,b,c,u et v                       */
}                           /*   fin de la définition de f1                 */
```

```
int main()                    /*   début du main, il n'a pas de paramètre     */
{
...                           /*   instructions qui appellerons p1 et f1      */
}                             /*   fin de la définition de main               */
```

1.18 Terminaison d'un programme

On peut réaliser la terminaison d'un programme en appelant la fonction `exit` qui fait partie de la bibliothèque standard. Cette fonction admet un paramètre entier qui est un code indiquant le type de la terminaison : une valeur nulle indique que le programme s'est convenablement terminé, toute valeur non nulle indique une terminaison avec erreur. Cette valeur est transmise à l'environnement d'exécution du programme [7] d'une manière dépendante de l'implémentation.

Un retour de la fonction `main` est équivalent à un appel à la fonction `exit` en lui passant en paramètre la valeur retournée par `main`. Si l'environnement d'exécution teste la valeur rendue par le programme, il faut donc terminer la fonction `main` par `return(0)`.

1.19 Mise en œuvre du compilateur C sous UNIX

Le lecteur sera supposé maîtriser un éditeur de textes lui permettant de créer un fichier contenant le source d'un programme. Supposons que le nom d'un tel fichier soit `essai1.c`, pour en réaliser sous UNIX la compilation et l'édition de liens avec la bibliothèque standard, il faut émettre la commande :

```
cc -o essai1 essai1.c
```

le binaire exécutable se trouve alors dans le fichier `essai1`. Pour l'exécuter, il suffit d'émettre la commande :

```
essai1
```

Pour vérifier qu'il n'y a pas de problème pour la mise en œuvre du compilateur, on peut essayer sur un des plus petits programmes possibles, à savoir un programme sans variables globales, et n'ayant qu'une seule procédure qui sera la procédure main. Exemple :

```
#include <stdio.h>
int main()
{
printf("ça marche!!\n");
}
```

1.20 Exercice

Écrire un programme comportant :

1. la déclaration de 3 variables globales entières heures, minutes, secondes ;

7. Sous UNIX ce sera le *shell* de l'utilisateur

2. une procédure `print_heure` qui imprimera le message :

   ```
   Il est ... heure(s) ... minute(s) ... seconde(s)
   ```

   en respectant l'orthographe du singulier et du pluriel ;

3. une procédure `set_heure` qui admettra trois paramètres de type entiers h, m, s, dont elle affectera les valeurs respectivement à heures, minutes et secondes ;

4. une procédure `tick` qui incrémentera l'heure de une seconde ;

5. la procédure `main` sera un jeu d'essai des procédures précédentes ;

Une solution possible est donnée ci-après.

```
#include <stdio.h>
int heures, minutes, secondes;

/*****************************************************************************/
/*                                                                           */
/*                              print_heure                                  */
/*                                                                           */
/*   But:                                                                    */
/*      Imprime l'heure                                                      */
/*                                                                           */
/*   Interface:                                                              */
/*      Utilise les variables globales heures, minutes, secondes             */
/*                                                                           */
/*****************************************************************************/
void print_heure()
{
printf("Il est %d heure",heures);
if (heures > 1) printf("s");
printf(" %d minute",minutes);
if (minutes > 1) printf("s");
printf(" %d seconde",secondes);
if (secondes > 1) printf("s");
printf("\n");
}

/*****************************************************************************/
/*                                                                           */
/*                              set_heure                                    */
/*                                                                           */
/*   But:                                                                    */
/*      Met l'heure à une certaine valeur                                    */
/*                                                                           */
/*   Interface:                                                              */
/*      h, m, s sont les valeurs à donner à heures, minutes, secondes        */
/*                                                                           */
/*****************************************************************************/
void set_heure(int h, int m, int s)
{
heures = h; minutes = m; secondes = s;
}

/*****************************************************************************/
/*                                                                           */
/*                              tick                                         */
/*                                                                           */
/*   But:                                                                    */
/*      Incrémente l'heure de une seconde                                    */
/*                                                                           */
/*   Interface:                                                              */
/*      Utilise les variables globales heures, minutes, secondes             */
/*                                                                           */
/*****************************************************************************/
void tick()
```

```
{
secondes = secondes + 1;
if (secondes >= 60)
   {
   secondes = 0;
   minutes = minutes + 1;
   if (minutes >= 60)
      {
      minutes = 0;
      heures = heures + 1;
      if (heures >= 24) heures = 0;
      }
   }
}

/******************************************************************************/
/*                                                                            */
/*                              main                                          */
/*                                                                            */
/******************************************************************************/
int main()
{
set_heure(3,32,10);
tick();
print_heure();

set_heure(1,32,59);
tick();
print_heure();

set_heure(3,59,59);
tick();
print_heure();

set_heure(23,59,59);
tick();
print_heure();
}
```

1.21 Récréation

Ami lecteur, si vous m'avez suivi jusqu'au bout de ce chapitre indigeste, vous méritez un divertissement.

Tous les ans, est organisé sur Internet le concours international du code C le plus obscur (International Obfuscated C Code Competition, IOCCC en abrégé). Le but est de produire un programme qui se compile et s'exécute sans erreur, dont le source est volontairement le plus obscur possible (ce qui est facile), mais qui est « intéressant » à un titre ou à un autre. C'est sur ce dernier point que doit s'exercer la créativité des participants. Tous les ans, les meilleures contributions sont de véritables joyaux et sont librement accessibles via l'URL : `ftp://ftp.uu.net/pub/ioccc`.

Nous présentons ci-dessous la contribution de Brian Westley pour l'année 1990. Son programme a ceci d'extraordinaire que le source C peut se lire comme un texte en « anglais » : il s'agit d'une conversation épistolaire entre un amoureux et sa belle (rétive, hélas pour lui). Il y a quelques petites licences prises par rapport à du vrai anglais : `1s` est pris comme une approximation du mot anglais *is*, `1l` est pris comme approximation de *ll* (dans I'll get a break) et des signes cabalistiques sont introduits entre les mots, mais c'est parfaitement lisible.

Il y a aussi un problème avec le langage C : le compilateur utilisé en 1990 par B. Westley n'était pas ANSI et acceptait le suffixe `s` derrière une constante entière pour lui donner le type `short int`. Si on transforme les `1s` en `1`, le programme se compile et s'exécute sans erreur. L'exécutable est un programme ... d'effeuillage de la marguerite ! Il faut lui passer en paramètre un nombre sensé être le nombre de pétales de la marguerite, et le programme va « dire » : *love me*, *love me not*, *love me*, *love me not* etc.[8] un nombre de fois égal au nombre de pétales. Voici le texte du programme :

```
char*lie;

double time, me= !0XFACE,
not; int rested,   get, out;
main(ly, die) char ly, **die ;{

    signed char lotte,

dear; (char)lotte--;
for(get= !me;; not){
1 -  out & out ;lie;{
char lotte, my= dear,
**let= !!me *!not+ ++die;

    (char*)(lie=
```

8. Les anglais semblent avoir une conception binaire du sentiment amoureux

```
"The gloves are OFF this time, I detest you, snot\n\0sed GEEK!");

do {not= *lie++ & 0xF00L* !me;
#define love (char*)lie -
love 1s *!(not= atoi(let
[get -me?

    (char)lotte-

(char)lotte: my- *love -
'I'  -  *love -  'U' -
'I'  -  (long)  - 4 - 'U' ])- !!
(time  =out=  'a'));} while( my - dear
&& 'I'-1l  -get-  'a'); break;}}

    (char)*lie++;

(char)*lie++, (char)*lie++; hell:0, (char)*lie;
get *out* (short)ly   -0-'R'-  get- 'a'^rested;
do {auto*eroticism,
that; puts(*( out
    - 'c'
-('P'-'S') +die+ -2 ));}while(!"you're at it");

for (*((char*)&lotte)^=
(char)lotte; (love ly) [(char)++lotte+
!!0xBABE];){ if ('I' -lie[ 2 +(char)lotte]){ 'I'-1l ***die; }
else{ if ('I' * get *out* ('I'-1l **die[ 2 ])) *((char*)&lotte) -=
'4' - ('I'-1l); not; for(get=!

get; !out; (char)*lie  &  0xD0- !not) return!!
(char)lotte;}

(char)lotte;
do{ not* putchar(lie [out
*!not* !!me +(char)lotte]);
not; for(;!'a';);}while(

    love (char*)lie);{

register this; switch( (char)lie
[(char)lotte] -1s *!out) {
char*les, get= 0xFF, my; case' ':
*((char*)&lotte) += 15; !not +(char)*lie*'s';
this +1s+ not; default: 0xF +(char*)lie;}}}
get - !out;
if (not--)
goto hell;

    exit( (char)lotte);}
```

Chapitre 2

Les tableaux

Dans ce chapitre nous allons voir comment déclarer un tableau, comment l'initialiser, comment faire référence à un des ses éléments. Du point de vue algorithmique, quand on utilise des tableaux, on a besoin d'intructions itératives, nous passerons donc en revue l'ensemble des instructions itératives du langage. Nous terminerons par un certain nombre d'opérateurs très utiles pour mettre en œuvre ces instructions.

2.1 Les tableaux

2.1.1 Déclaration de tableaux dont les éléments ont un type de base

Pour déclarer un tableau dont les éléments ont un type de base :

- partir de la déclaration de variable ayant un type de base ;
- ajouter entre crochets le nombre d'éléments du tableau après le nom.

Exemple :

```
int t[10];                  /*   t tableau de 10 int          */
long int t1[10], t2[20];    /*   t1 tableau de 10 long int,
                                 t2 tableau de 20 long int    */
```

En pratique, il est recommandé de toujours donner un nom à la constante qui indique le nombre d'éléments d'un tableau. Exemple :

```
#define N 100
int t[N];
```

Les points importants sont les suivants :

- les index des éléments d'un tableau vont de 0 à N - 1 ;
- la taille d'un tableau doit être connue statiquement par le compilateur.

Impossible donc d'écrire :

```
int t[n];
```

où `n` serait une variable.

2.1.2 Initialisation d'un tableau

Il est possible d'initialiser un tableau avec une liste d'expressions constantes séparées par des virgules, et entourée des signes `{` et `}`. Exemple :

```
#define N 5
int t[N] = {1, 2, 3, 4, 5};
```

On peut donner moins d'expressions constantes que le tableau ne comporte d'éléments. Dans ce cas, les premiers éléments du tableau seront initialisés avec les valeurs indiquées, les autres seront initialisés à zéro. Exemple :

```
#define N 10
int t[N] = {1, 2};
```

Les éléments d'indice 0 et 1 seront initialisés respectivement avec les valeurs 1 et 2, les autres éléments seront initialisés à zéro.

Il n'existe malheureusement pas de facteur de répétition, permettant d'exprimer « initialiser n éléments avec la même valeur v ». Il faut soit mettre n fois la valeur v dans l'initialisateur, soit initialiser le tableau par des instructions.

Cas particulier des tableaux de caractères

- Un tableau de caractères peut être initialisé par une liste de constantes caractères. Exemple :

  ```
  char ch[3] = {'a', 'b', 'c'};
  ```

  C'est évidemment une méthode très lourde.

- Un tableau de caractères peut être initialisé par une chaîne littérale. Exemple :

  ```
  char ch[8] = "exemple";
  ```

  On se rappelle que le compilateur complète toute chaîne littérale avec un caractère *null*, il faut donc que le tableau ait au moins un élément de plus que le nombre de caractères de la chaîne littérale.

- Il est admissible que la taille déclarée pour le tableau soit supérieure à la taille de la chaîne littérale. Exemple :

  ```
  char ch[100] = "exemple";
  ```

  dans ce cas, seuls les 8 premiers caractères de `ch` seront initialisés.

- Il est également possible de ne pas indiquer la taille du tableau et dans ce cas, le compilateur a le bon goût de compter le nombre de caractères de la chaîne littérale et de donner la taille adéquate au tableau (sans oublier le *null*). Exemple :

  ```
  char ch[] = "ch aura 22 caractères";
  ```

- Il est également possible de donner au tableau une taille égale au nombre de caractères de la chaîne. Dans ce cas, le compilateur comprend qu'il ne faut pas rajouter le *null* de la fin de chaîne. Exemple :

  ```
  char ville[8] = "bordeaux";
  ```

2.1.3 Référence à un élément d'un tableau

- Syntaxe :

  Dans sa forme la plus simple, une référence à un élément de tableau a la syntaxe suivante :

  expression :
  $\Rightarrow$ *nom-de-tableau* `[` $expression_1$ `]`

- Sémantique : $expression_1$ doit délivrer une valeur entière, et l'*expression* délivre l'élément d'indice $expression_1$ du tableau. Une telle *expression* est une *lvalue*, on peut donc la rencontrer aussi bien en partie gauche qu'en partie droite d'affectation.

Par exemple, dans le contexte de la déclaration :

```
#define N 10
int t[N];
```

on peut écrire :

```
x = t[i];       /*  référence à l'élément d'indice i du tableau t        */
t[i+j] = k;     /*  affectation de l'élément d'indice i+j du tableau t   */
```

2.1.4 Chaînes et tableaux de caractères

Le langage C fait la distinction entre tableau de caractères et chaîne de caractères : une chaîne de caractères est un tableau de caractères dont la fin est indiquée par un caractère *null*. C'est une convention très pratique exploitée par la bibliothèque standard.

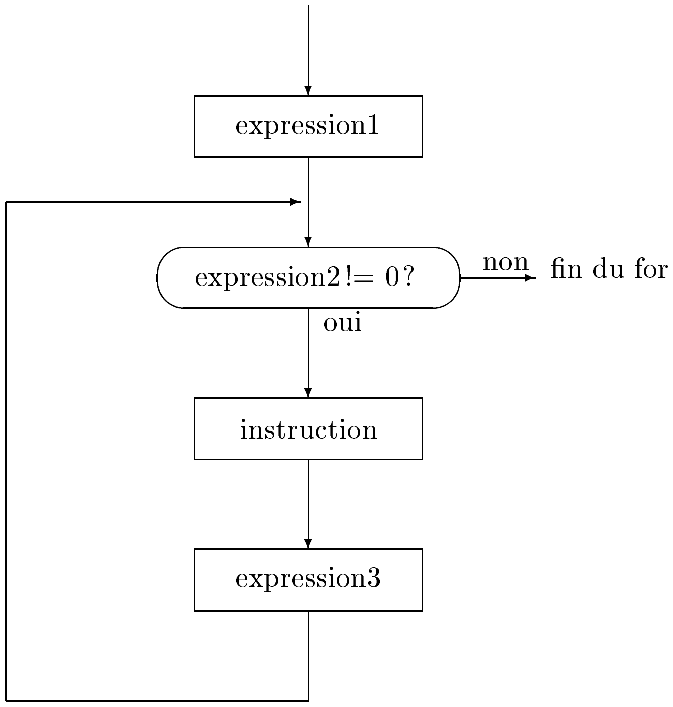
2.2 Les instructions itératives

2.2.1 Instruction for

- Syntaxe :

  instruction :
  $\Rightarrow$ `for` `(` $expression_{1\ option}$ `;` $expression_{2\ option}$ `;` $expression_{3\ option}$ `)` *instruction*

- Sémantique : l'exécution réalisée correspond à l'organigramme suivant :



Lorsque l'on omet *expression*$_1$ et/ou *expression*$_2$ et/ou *expression*$_3$, la sémantique est celle de l'organigramme précédent, auquel on a enlevé la ou les parties correspondantes.

Remarques

On voit que la vocation de *expression*$_1$ et *expression*$_3$ est de réaliser des effets de bord, puisque leur valeur est inutilisée. Leur fonction logique est d'être respectivement les parties initialisation et itération de la boucle. L' *expression*$_2$ est utilisée pour le test de bouclage. L'*instruction* est le travail de la boucle.

Exemple de boucle `for`

Initialisation d'un tableau à zéro :

```
#define N 10
int t[N];
for (i = 0; i < N; i = i + 1) t[i] = 0;
```

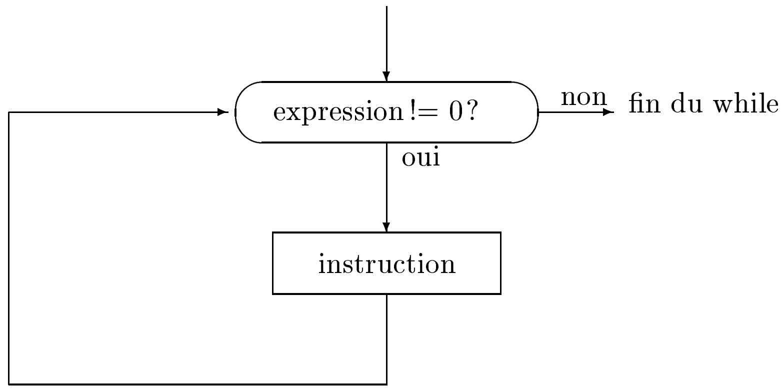
2.2.2 Instruction `while`

- Syntaxe :

  instruction :
  ⇒ `while` `(` *expression* `)` *instruction*

- Sémantique :

  l'exécution réalisée correspond à l'organigramme suivant :



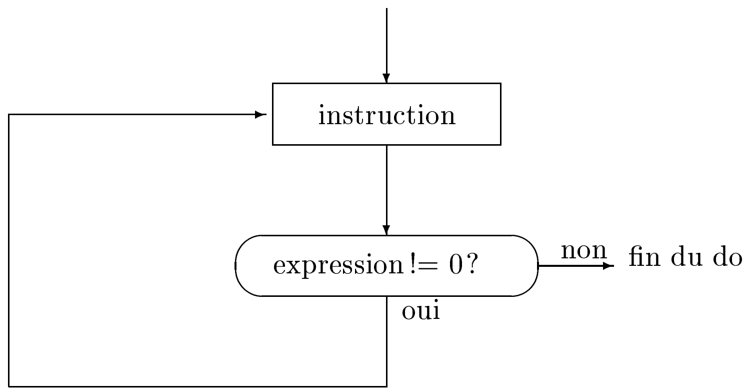
2.2.3 Instruction `do`

- Syntaxe :

  instruction :
  $\Rightarrow$ `do` *instruction* `while` `(` *expression* `)` `;`

- Sémantique :

  l'exécution réalisée correspond à l'organigramme suivant :



2.2.4 Instruction `break`

- Syntaxe :

  instruction :
  $\Rightarrow$ `break` `;`

- Sémantique :

  Provoque l'arrêt de la première instruction `for`, `while`, `do` englobante.

Exemple

L'instruction `for` ci-dessous est stoppée au premier `i` tel que `t[i]` est nul :

```
for (i = 0; i < N; i = i + 1)
   if (t[i] == 0) break;
```

2.2.5 Instruction `continue`

- Syntaxe :

  instruction :

  ⇒ `continue ;`

- Sémantique :

  Dans une instruction `for`, `while` ou `do`, l'instruction `continue` provoque l'arrêt de l'itération courante, et le passage au début de l'itération suivante.

Exemple

Supposons que l'on parcoure un tableau `t` pour réaliser un certain traitement sur tous les éléments, sauf ceux qui sont négatifs :

```
for (i = 0; i < N; i = i + 1)
   {
   if (t[i] < 0 ) continue;   /*  on passe au i suivant dans le for   */
   ...                        /*  traitement de l'élément courant     */
   }
```

2.3 Les opérateurs

2.3.1 Opérateur pré et postincrément

Le langage C offre un opérateur d'incrémentation qui peut être utilisé soit de manière préfixé, soit de manière postfixé. Cet opérateur se note `++` et s'applique à une *lvalue*. Sa syntaxe d'utilisation est donc au choix, soit `++` *lvalue* (utilisation en préfixé), soit *lvalue* `++` (utilisation en postfixé). Tout comme l'opérateur d'affectation, l'opérateur d'incrémentation réalise à la fois un effet de bord et délivre une valeur :
`++` *lvalue* incrémente lvalue de 1 et délivre cette nouvelle valeur.
lvalue `++` incrémente lvalue de 1 et délivre **la valeur initiale** de *lvalue*.

Exemples

Soient `i` un `int` et `t` un tableau de `int` :

```
i = 0;
t[i++] = 0;  /*  met à zéro l'élément d'indice 0  */
t[i++] = 0;  /*  met à zéro l'élément d'indice 1  */

i = 1;
```

```
t[++i] = 0;  /*  met à zéro l'élément d'indice 2  */
t[++i] = 0;  /*  met à zéro l'élément d'indice 3  */
```

2.3.2 Opérateur pré et postdécrément

Il existe également un opérateur de décrémentation qui partage avec l'opérateur incrément les caractéristiques suivantes :

- il peut s'utiliser en préfixe ou en postfixé ;
- il s'applique à une *lvalue,* ;
- il fait un effet de bord et délivre une valeur.

Cet opérateur se note `--` et décrémente la *lvalue* de 1 :
`--` *lvalue* décrémente lvalue de 1 et délivre cette nouvelle valeur.
lvalue `--` décrémente lvalue de 1 et délivre **la valeur initiale** de lvalue.

Exemples

Soient `i` un `int` et `t` un tableau de `int` :

```
i = 9;
t[i--] = 0;  /*  met à zéro l'élément d'indice 9  */
t[i--] = 0;  /*  met à zéro l'élément d'indice 8  */

i = 8;
t[--i] = 0;  /*  met à zéro l'élément d'indice 7  */
t[--i] = 0;  /*  met à zéro l'élément d'indice 6  */
```

2.3.3 Quelques utilisations typiques de ces opérateurs

Utilisation dans les instructions *expression*

On a vu qu'une des formes d'instruction possibles en C est :
expression ;
et que cela n'a de sens que si l'*expression* réalise un effet de bord. Les opérateurs `++` et `--` réalisant précisément un effet de bord, permettent donc d'écrire des instructions se réduisant à une expression utilisant un de ces opérateurs. Une incrémentation ou une décrémentation de variable se fait classiquement en C de la manière suivante :

```
i++;    /*   incrémentation de i   */
j--;    /*   décrémentation de j   */
```

Utilisation dans les instructions itératives

Une boucle `for` de parcours de tableau s'écrit typiquement de la manière suivante :

```
for (i = 0; i < N; i++)
   {
   ...
   }
```

2.3.4 Opérateur *et logique*

- Syntaxe :

  expression :
  $\Rightarrow$ $expression_1$ `&&` $expression_2$

- Sémantique :

  expression_1 est évaluée et :

  1. si sa valeur est nulle, l'expression `&&` rend la valeur 0 ;
  2. si sa valeur est non nulle, expression_2 est évaluée, et l'expression `&&` rend la valeur 0 si expression_2 est nulle, et 1 sinon.

On voit donc que l'opérateur `&&` réalise le *et logique* de $expression_1$ et $expression_2$ (en prenant pour faux la valeur 0, et pour vrai toute valeur différente de 0).

Remarque sur la sémantique

On a la certitude que $expression_2$ ne sera pas évaluée si $expression_1$ rend la valeur faux. Ceci présente un intérêt dans certains cas de parcours de tableau ou de liste de blocs chaînés. Par exemple dans le cas d'un parcours de tableau à la recherche d'un élément ayant une valeur particulière, supposons que l'on utilise comme test de fin de boucle l'expression

```
i < n && t[i] != 234
```

(on boucle tant que ce test est vrai), où `i < n` est le test permettant de ne pas déborder du tableau, et `t[i] != 234` est le test de l'élément recherché. S'il n'existe dans le tableau aucun élément égal à 234, il va arriver un moment où on va évaluer `i < n && t[i] != 234` avec `i = n`. La sémantique de l'opérateur `&&` assurant que l'expression `t[i] != 234` ne sera pas évaluée, on ne court donc pas le risque d'avoir une erreur matérielle (`t[n]` peut référencer une adresse mémoire invalide).

Exemples d'utilisation

```
int a,b;
if (a > 32 && b < 64) ...
if ( a && b > 1)  ...
b = (a > 32 && b < 64);
```

2.3.5 Opérateur *ou logique*

- Syntaxe :

  expression :
  $\Rightarrow$ $expression_1$ `||` $expression_2$

- Sémantique :

  $expression_1$ est évaluée et :

  – si sa valeur est non nulle, l'expression `||` délivre la valeur 1 ;

– sinon, *expression*$_2$ est évaluée, si sa valeur est nulle, l'expression `||` délivre la valeur 0 sinon elle délivre la valeur 1.

On voit donc que l'opérateur `||` réalise le *ou logique* de ses opérandes, toujours avec les mêmes conventions pour les valeurs vrai et faux, à savoir 0 pour faux, et n'importe quelle valeur non nulle pour vrai. Dans ce cas également, on a la certitude que le second opérande ne sera pas évalué si le premier délivre la valeur vrai.

Exemples

```
int a,b;
if (a > 32 || b < 64) ...
if ( a || b > 1)  ...
b = (a > 32 || b < 64);
```

2.3.6 Opérateur *non logique*

- Syntaxe :

  expression :

  $\Rightarrow$! *expression*

- Sémantique :

  expression est évaluée, si sa valeur est nulle, l'opérateur ! délivre la valeur 1, sinon il délivre la valeur 0.

  Cet opérateur réalise le *non logique* de son opérande.

2.4 Exercice

Déclarer un tableau `nb_jour` qui doit être initialisé de façon à ce que `nb_jour[i]` soit égal au nombre de jours du i^{eme} mois de l'année pour i allant de 1 à 12 (`nb_jour[0]` sera inutilisé).

Écrire une procédure d'initialisation de `nb_jour` qui utilisera l'algorithme suivant :

– si `i` vaut 2 le nombre de jours est 28 ;

– sinon si `i` pair et `i <= 7` ou `i` impair et `i > 7` le nombre de jours est 30 ;

– sinon le nombre de jours est 31.

Écrire une procédure d'impression des 12 valeurs utiles de `nb_jour`. La procédure `main` se contentera d'appeler les procédures d'initialisation et d'impression de `nb_jour`.

```
#include <stdio.h>
int nb_jours[13];

/*****************************************************************************/
/*                                                                           */
/*                              init_nb_jours                                */
/*                                                                           */
/*   But:                                                                    */
/*      Initialise le tableau nb_jours                                       */
/*                                                                           */
/*****************************************************************************/
void init_nb_jours()
{
int i;

for (i = 1; i <= 12; i++)
   if (i == 2)
      nb_jours[2] = 28;
   else if ( (i % 2 == 0) && i <= 7 || (i % 2 == 1) && i > 7 )
      nb_jours[i] = 30;
   else nb_jours[i] = 31;
}

/*****************************************************************************/
/*                                                                           */
/*                              print_nb_jours                               */
/*                                                                           */
/*   But:                                                                    */
/*      Imprime le contenu du tableau nb_jours                               */
/*                                                                           */
/*****************************************************************************/
void print_nb_jours()
{
int i;

for (i = 1; i <= 12; i++)
   printf("%d  ",nb_jours[i]);
printf("\n");
}

/*****************************************************************************/
/*                                                                           */
/*                              main                                         */
/*                                                                           */
/*****************************************************************************/
int main()
{
init_nb_jours();
print_nb_jours();
}
```

Chapitre 3

Les pointeurs

3.1 Notion de pointeur

Une valeur de type pointeur repère une variable. En pratique, cela signifie qu'une valeur de type pointeur est l'adresse d'une variable.

3.2 Déclarations de variables de type pointeur vers les types de base

Pour déclarer une variable pointeur vers un type de base :

- partir de la déclaration d'une variable ayant un type de base ;
- ajouter le signe * devant le nom de la variable.

Exemple :

```
int *pi;          /*   pi est un pointeur vers un int                 */
short int *psi;   /*   psi est un pointeur vers un short int          */
double *pd;       /*   pd pointeur vers un flottant double précision  */
char *pc;         /*   pc pointeur vers un char                       */
```

3.3 Type de pointeur générique

Le type `void *` est le type pointeur générique, c'est à dire capable de pointer vers n'importe quel type d'objet. Sans un tel type, il ne serait pas possible par exemple d'indiquer le type d'objet rendu par les fonctions d'allocation de mémoire qui rendent un pointeur vers l'objet alloué, puisque ce type varie d'une invocation à l'autre de la fonction.

Par exemple, la fonction `malloc` de la bibliothèque standard est définie de la manière suivante : `void *malloc(size_t size);` [1]

Note

La solution quand on utilisait les compilateurs K&R était d'utiliser le type `char *` qui jouait de manière conventionnelle ce rôle de pointeur générique. Le lecteur ne s'étonnera donc pas de trouver dans les vieux (?) manuels, la fonction `malloc` définie de la manière suivante :

```
char *malloc(size);
```

3.4 Opérateur adresse de

L'opérateur `&` appliqué à une variable délivre l'adresse de celle-ci ; cette adresse pourra être affectée à une variable de type pointeur. On peut écrire par exemple :

```
int i;
int *pi;   /*   pi pointeur vers un int   */

pi = &i;   /*   le pointeur pi repère la variable i   */
```

3.5 Opérateur d'indirection

Lorsque l'opérateur `*` est utilisé en opérateur préfixé, il agit de l'opérateur indirection qui, appliqué à une valeur de type pointeur, délivre la valeur pointée. On peut écrire par exemple :

```
int i;
int *pi;

pi = &i;        /*   initialisation du pointeur pi                 */
*pi = 2;        /*   initialisation de la valeur pointée par pi    */
j = *pi + 1;    /*   une utilisation de la valeur pointée par pi   */
```

Remarque

L'opérateur `*` est surchargé : il peut être opérateur de multiplication ou opérateur d'indirection. La grammaire lève l'ambiguïté car l'opérateur d'indirection est préfixé, alors que l'opérateur de multiplication est infixé. Surcharger les opérateurs gêne la lisibilité des programmes. Exemple : si `i` et `j` sont des pointeurs vers des entier, la multiplication des deux valeurs pointées s'écrit : `*i**j`

1. `size_t` est un type défini dans `stddef.h`

Devinette

Si `i` et `j` sont des pointeurs vers des entiers, `*i**j` est le produit des valeurs pointées, mais `*i/*j` est-il le quotient des valeurs pointées ? Réponse à la fin de ce chapitre.

3.6 Exercice

1. Déclarer un entier `i` et un pointeur `p` vers un entier ;
2. Initialiser l'entier à une valeur arbitraire et faire pointer `p` vers `i` ;
3. Imprimer la valeur de `i` ;
4. Modifier l'entier pointé par `p` (en utilisant `p`, pas `i`) ;
5. Imprimer la valeur de `i`.

Une solution possible est donnée page suivante.

```
#include <stdio.h>

/*****************************************************************************/
/*                                                                           */
/*                              main                                         */
/*                                                                           */
/*****************************************************************************/
int main()
{
int i;
int *p;

i = 1;
p = &i;

printf("valeur de i avant: %d\n",i);
*p = 2;
printf("valeur de i après: %d\n",i);
}
```

3.7 Pointeurs et opérateurs additifs

3.7.1 Opérateurs + et -

L'opérateur + permet de réaliser la somme de deux valeurs arithmétiques, mais il permet également de réaliser la somme d'un pointeur et d'un entier. Une telle opération n'a de sens cependant, que si le pointeur repère un élément d'un tableau.

Soient p une valeur pointeur vers des objets de type T et un tableau dont les éléments sont du même type T ; si p repère l'élément d'indice i du tableau, p + j **est une valeur de type pointeur vers T**, qui repère l'élément d'indice i + j du tableau (en supposant qu'il existe).

Il en va de même avec l'opérateur soustraction : si p repère l'élément d'indice i d'un tableau, p - j repère l'élément d'indice i - j du tableau (toujours en supposant qu'il existe). Exemple :

```
#define N 10
int t[N];
int *p,*q,*r,*s;

p = &t[0];        /*   p repère le premier élément de t   */
q = p + (N-1);    /*   q repère le dernier élément de t   */

r = &t[N-1];      /*   r repère le dernier élément de t   */
s = r - (N-1);    /*   s repère le premier élément de t   */
```

La norme précise que pour réaliser la somme ou la différence d'un pointeur et d'un entier, il faut qu'à la fois le pointeur et le résultat repèrent les éléments d'un même tableau ou l'élément (fictif) après le dernier élément du tableau. En d'autre termes, si on a :

```
#define N 100
int t[N];
int * p = &t[0];
```

L'expression `p + N` est valide, mais `p - 1` ou `p + N + 1` ne sont pas valides. La possibilité de référencer l'élément (fictif) après le dernier élément d'un tableau a été introduite pour les problèmes de fin de boucle de parcours de tableau (mais on aurait pu s'en passer).

3.7.2 Opérateurs ++ et --

On peut appliquer les opérateurs `++` et `--` à des pointeurs et il est classique de les utiliser pour réaliser des parcours de tableaux. Exemple (on rappelle que toute chaîne est terminée par un *null*, c'est à dire le caractère `'\0'`) :

```
char mess[] = "Hello world!!";
char *p;

for (p = &mess[0]; *p != '\0'; p++)
   {
   /*   ici p repère l'élément courant de mess   */
   }
```

Autre classique, en reprenant les variables `mess` et `p` de l'exemple précédent :

```
p = &mess[0];
while (*p != '\0')
   {
   /*   ici utilisation de *p++   */
   }
```

3.8 Différence de deux pointeurs

Il est possible d'utiliser l'opérateur de soustraction pour calculer la différence de deux pointeurs. Cela n'a de sens que si les deux pointeurs repèrent des éléments d'un même tableau.

Soient p1 et p2 deux pointeurs du même type tels que p1 repère le i^{eme} élément d'un tableau, et p2 repère le j^{eme} élément du même tableau, p2 - p1 **est une valeur de type `ptrdiff_t`** qui est égale à j - i. Le type `ptrdiff_t` est défini dans le fichier d'include `stddef.h` En pratique, une variable de type `ptrdiff_t` pourra être utilisée comme une variable de type `int`. La norme précise qu'il est valide de calculer la différence de deux pointeurs à condition que tous deux repèrent des éléments d'un même tableau, ou l'élément (fictif) après le dernier élément du tableau.

3.9 Exercice

Déclarer et initialiser statiquement un tableau d'entiers `t` avec des valeurs dont certaines seront nulles. Écrire une procédure qui parcoure le tableau `t` et qui imprime les index des éléments nuls du tableau, **sans utiliser aucune variable de type entier**. Une solution possible est donnée page suivante.

```c
#include <stdio.h>

#define N 10
int t[N] = {1,2,0,11,0,12,13,14,0,4};

/*****************************************************************************/
/*                                                                           */
/*                                print1                                     */
/*   Premiere version                                                        */
/*                                                                           */
/*****************************************************************************/
void print1()
{
int *pdeb,*pfin,*p;

pdeb = &t[0];      /*   repère le premier élément de t   */
pfin = &t[N-1];    /*   repère le dernier élément de t   */

for (p = pdeb; p <= pfin; p++)
   if (*p == 0) printf("%d ",p - pdeb);
printf("\n");
}

/*****************************************************************************/
/*                                                                           */
/*                                print2                                     */
/*   Une autre version                                                       */
/*                                                                           */
/*****************************************************************************/
void print2()
{
int *pdeb,*pfin,*p;

pdeb = &t[0];     /* repère le premier élément de t                          */
pfin = pdeb + N;  /* repère l'élément (fictif) apres le dernier élément de t */

for (p = pdeb; p < pfin; p++)
   if (*p == 0) printf("%d ",p - pdeb);
printf("\n");
}

/*****************************************************************************/
/*                                                                           */
/*                                main                                       */
/*                                                                           */
/*****************************************************************************/
int main()
{
print1();
print2();
}
```

3.10 Passage de paramètres

3.10.1 Les besoins du programmeur

En ce qui concerne le passage de paramètres à une procédure, le programmeur a deux besoins fondamentaux :

- soit il désire passer une valeur qui sera exploitée par l'algorithme de la procédure (c'est ce dont on a besoin quand on écrit par exemple sin(x)). Une telle façon de passer un paramètre s'appelle du *passage par valeur* ;

- soit il désire passer une référence à une variable, de manière à permettre à la procédure de modifier la valeur de cette variable. C'est ce dont on a besoin quand on écrit une procédure réalisant le produit de deux matrices `prodmat(a,b,c)` où l'on veut qu'en fin d'exécution de `prodmat`, la matrice `c` soit égale au produit matriciel des matrices `a` et `b`. `prodmat` a besoin des valeurs des matrices `a` et `b`, et d'une référence vers la matrice `c`. Une telle façon de passer un paramètre s'appelle du *passage par adresse.*

3.10.2 Comment les langages de programmation satisfont ces besoins

Face à ces besoins, les concepteurs de langages de programmation ont imaginé différentes manières de les satisfaire, et quasiment chaque langage de programmation dispose de sa stratégie propre de passage de paramètres.

- Une première possibilité consiste à arguer du fait que le passage de paramètre par adresse est plus puissant que le passage par valeur, et à réaliser tout passage de paramètre par adresse (c'est la stratégie de FORTRAN et PL/1) ;

- Une seconde possibilité consiste à permettre au programmeur de déclarer explicitement quels paramètres il désire passer par valeur et quels paramètres il désire passer par adresse (c'est la stratégie de PASCAL) ;

- La dernière possibilité consistant à réaliser tout passage de paramètre par valeur semble irréaliste puisqu'elle ne permet pas de satisfaire le besoin de modification d'un paramètre. C'est cependant la stratégie choisie par les concepteurs du langage C.

3.10.3 La stratégie du langage C

En C, tout paramètre est passé par valeur, et cette règle ne souffre aucune exception. Cela pose le problème de réaliser un passage de paramètre par adresse lorsque le programmeur en a besoin. La solution à ce problème consiste dans ce cas, à déclarer le paramètre comme étant un pointeur. Cette solution n'est rendue possible que par l'existence de l'opérateur *adresse de* qui permet de délivrer l'adresse d'une *lvalue.*

Voyons sur un exemple. Supposons que nous désirions écrire une procédure `add`, admettant trois paramètres `a`, `b` et `c`. Nous désirons que le résultat de l'exécution de `add` soit d'affecter au paramètre `c` la somme des valeurs des deux premiers paramètres. Le paramètre `c` ne peut évidemment pas être passé par valeur, puisqu'on désire modifier la valeur du paramètre effectif correspondant. Il faut donc programmer `add` de la manière suivante :

```
void add(int a, int b, int *c)
/*   c repère l'entier où on veut mettre le résultat   */

{
*c = a + b;
}

int main()
{
int i,j,k;

/*   on passe les valeurs de i et j comme premiers paramètres  */
/*   on passe l'adresse de k comme troisième paramètre         */
add(i,j,&k);
}
```

3.11 Discussion

1. Nous estimons qu'il est intéressant pour le programmeur de raisonner en terme de *passage par valeur* et de *passage par adresse* et qu'il est préférable d'affirmer, lorsque l'on écrit `f(&i);` « `i` est passé par adresse à `f` », plutôt que d'affirmer « l'adresse de `i` est passée par valeur à `f` », tout en sachant que c'est la deuxième affirmation qui colle le mieux à la stricte réalité du langage. Que l'on ne s'étonne donc pas dans la suite de ce manuel de nous entendre parler de passage par adresse.

2. Nous retiendrons qu'en C, le passage de paramètre par adresse est entièrement géré par le programmeur. C'est à la charge du programmeur de déclarer le paramètre concerné comme étant de type pointeur vers ... et de bien songer, lors de l'appel de la fonction, à passer l'adresse du paramètre effectif.

3.12 Une dernière précision

Quand un langage offre le passage de paramètre par valeur, il y a deux possibilités :

1. soit le paramètre est une constante (donc non modifiable)

2. soit le paramètre est une variable locale à la procédure. Cette variable est initialisée lors de l'appel de la procédure avec la valeur du paramètre effectif.

C'est la seconde solution qui a été retenue par les concepteurs du langage C. Voyons sur un exemple. Supposons que l'on désire écrire une fonction `sum` admettant comme paramètre `n` et qui rende la somme des `n` premiers entiers. On peut programmer de la manière suivante :

```
int sum(int n)
{
int r = 0;

for ( ; n > 0; n--) r = r + n;
return(r);
}
```

On voit que le paramètre `n` est utilisé comme variable locale, et que dans l'instruction `for`, la partie initialisation est vide puisque `n` est initialisée par l'appel de `sum`.

3.13 Exercice

On va coder un algorithme de cryptage très simple : on choisit un décalage (par exemple 5), et un a sera remplacé par un f, un b par un g, un c par un h, etc. On ne cryptera que les lettres majuscules et minuscules sans toucher ni à la ponctuation ni à la mise en page (caractères blancs et *line feed*). On supposera que les codes des lettres se suivent de a à z et de A à Z. On demande de :

1. déclarer un tableau de caractères `mess` initialisé avec le message en clair ;
2. écrire une procédure `crypt` de cryptage d'un caractère qui sera passé par adresse ;
3. écrire le `main` qui activera `crypt` sur l'ensemble du message et imprimera le résultat.

```c
#include <stdio.h>

char mess[] = "Les sanglots longs des violons de l’automne\n\
blessent mon coeur d’une langeur monotone";

#define DECALAGE 5

/*****************************************************************************/
/*                                                                           */
/*                              crypt                                        */
/*                                                                           */
/*   But:                                                                    */
/*      Crypte le caractère passé en paramètre                               */
/*                                                                           */
/*   Interface:                                                              */
/*      p : pointe le caractère à crypter                                    */
/*                                                                           */
/*****************************************************************************/
void crypt(char *p)
{
enum {BAS, HAUT};
int casse;

if (*p >= ’a’ && *p <= ’z’) casse = BAS;
else if (*p >= ’A’ && *p <= ’Z’) casse = HAUT;
else return;

*p = *p + DECALAGE;
if (casse == BAS && *p > ’z’ || casse == HAUT && *p > ’Z’) *p = *p -26;
}

/*****************************************************************************/
/*                                                                           */
/*                              main                                         */
/*                                                                           */
/*****************************************************************************/
int main()
{
char *p;
int i;

/*   phase de cryptage   */
p = &mess[0];
while(*p)
   crypt(p++);

/*   impression du résultat   */
printf("Résultat:\n");
printf(mess);
printf("\n");
}
```

3.14 Lecture formattée

Il existe dans la bibliothèque standard une fonction de lecture formattée qui fonctionne selon le même principe que la procédure `printf`. Sa syntaxe d'utilisation est la suivante :
`scanf ( ` *format* ` , ` *liste-d'expressions* ` ) ;`
format est une chaîne de caractères indiquant sous forme de séquences d'échappement les entités que `scanf` lit sur l'*entrée standard* :

- `%d` pour un nombre décimal ;
- `%x` pour un nombre écrit en hexadécimal ;
- `%c` pour un caractère.

Tous les éléments de la *liste-d'expressions* doivent délivrer après évaluation l'adresse de la variable dans laquelle on veut mettre la valeur lue. Si le flot de caractères lu sur l'*entrée standard* ne correspond pas à *format*, `scanf` cesse d'affecter les variables de *liste-d'expressions* à la première disparité. Exemple :

```
scanf("%d %x",&i,&j);
```

cette instruction va lire un nombre écrit en décimal et mettre sa valeur dans la variable `i`, puis lire un nombre écrit en hexadécimal et mettre sa valeur dans la variable `j`. On aura remarqué que les paramètres `i` et `j` ont étés passés par adresse à `scanf`. Si l'*entrée standard* commence par un nombre décimal non suivi d'un nombre hexadécimal, seul `i` recevra une valeur. Si l'*entrée standard* ne commence pas par un nombre décimal, ni `i` ni `j` ne recevront de valeur.

Valeur rendue

Sur rencontre de fin de fichier, la fonction `scanf` rend `EOF`, sinon elle rend le nombre de variables qu'elle a pu affecter.

Attention

Une erreur facile à commettre est d'omettre les opérateurs `&` devant les paramètres de `scanf`. C'est une erreur difficile à détecter car le compilateur ne donnera aucun message d'erreur et à l'exécution, ce sont les valeurs de `i` et `j` qui seront interprétées comme des adresses par `scanf`. Avec un peu de chance ces valeurs seront des adresses invalides, et le programme s'avortera[2] sur l'exécution du `scanf`, ce qui donnera une idée du problème. Avec un peu de malchance, ces valeurs donneront des adresses parfaitement acceptables, et le `scanf` s'exécutera en allant écraser les valeurs d'autres variables qui ne demandaient rien à personne. Le programme pourra s'avorter beaucoup plus tard, rendant très difficile la détection de l'erreur.

3.15 Les dernières instructions

Le langage C comporte 3 instructions que nous n'avons pas encore vu : un *if* généralisé, un *goto* et une instruction nulle.

2. Le terme avorter est à prendre au sens technique de *abort*

3.15.1 Instruction `switch`

Le langage C offre une instruction `switch` qui est un `if` généralisé.

- Syntaxe :

  instruction :

  ⇒ `switch (` *expression* `)`

  `{`

  `case` *expression*$_1$ `:` *liste-d'instructions*$_{1\ option}$ `break;`$_{option}$

  `case` *expression*$_2$ `:` *liste-d'instructions*$_{2\ option}$ `break;`$_{option}$

  ....

  `case` *expression*$_n$ `:` *liste-d'instructions*$_{n\ option}$ `break;`$_{option}$

  `default :` *liste-d'instructions*

  `}`

  De plus :

  – toutes les *expression*$_i$ doivent délivrer une valeur connue à la compilation ;

  – il ne doit pas y avoir deux *expression*$_i$ délivrant la même valeur ;

  – l'alternative `default` est optionnelle.

- Sémantique :

  1. *expression* est évaluée, puis le résultat est comparé avec *expression*$_1$, *expression*$_2$, etc.
  2. à la première *expression*$_i$ dont la valeur est égale à celle de *expression*, on exécute la (ou les)[3] *liste-d'instructions* correspondante(s) jusqu'à la rencontre de la première instruction `break;`. La rencontre d'une instruction `break` termine l'exécution de l'instruction `switch`.
  3. si il n'existe aucune *expression*$_i$ dont la valeur soit égale à celle de *expression*, on exécute la *liste-d'instructions* de l'alternative `default` si celle-ci existe, sinon on ne fait rien.

Discussion

Vu le nombre de parties optionnelles dans la syntaxe, il y a 3 types d'utilisations possibles pour le `switch`. Première possibilité, on peut avoir dans chaque alternative une *liste-d'instructions* et un `break;` comme dans l'exemple suivant :

3. la ou les, car dans chaque `case`, le `break` est optionnel

```
enum {BLEU=1, BLANC, ROUGE};

void print_color(int color)
{
switch(color)
   {
   case BLEU : printf("bleu"); break;
   case BLANC : printf("blanc"); break;
   case ROUGE : printf("rouge"); break;
   default : printf("erreur interne du logiciel numéro xx\n");
   }
}
```

Deuxième possibilité, on peut avoir une ou plusieurs alternatives ne possédant ni *liste-d'instructions*, ni `break;`. Supposons que l'on désire compter dans une suite de caractères, le nombre de caractères qui sont des chiffres, et le nombre de caractères qui ne le sont pas. On peut utiliser le `switch` suivant :

```
switch(c)
   {
   case '0':
   case '1':
   case '2':
   case '3':
   case '4':
   case '5':
   case '6':
   case '7':
   case '8':
   case '9': nb_chiffres++; break;
   default: nb_non_chiffres++;
   }
```

Troisième possibilité, une alternative peut ne pas avoir de `break` comme dans l'exemple suivant :

```
enum {POSSIBLE, IMPOSSIBLE};

void print_cas(int cas)
{
switch (cas)
   {
   case IMPOSSIBLE: printf("im");
   case POSSIBLE:   printf("possible\n"); break;
   case default:    printf("erreur interne du logiciel numéro xx\n");
   }
}
```

Une telle utilisation du `switch` pose un problème de lisibilité, car l'expérience montre que l'absence du `break;` est très difficile à voir. Il est donc recommandé de mettre un commentaire, par exemple de la façon suivante :

```
case IMPOSSIBLE: printf("im");   /*   ATTENTION: pas de break;   */
```

Remarque

Le mot-clé `break` est surchargé : nous avons vu au chapitre 2.2.4 que l'instruction `break` permettait de stopper l'exécution d'une instruction itérative `for`, `while`, `do`. Il est utilisé ici de manière complètement différente.

3.15.2 Instruction `goto`

- Syntaxe :

  instruction :

  ⇒ `goto` *identificateur* ;

- Sémantique :

  Toute instruction peut être précédée d'un identificateur suivi du signe :. Cet identificateur est appelé *étiquette*. Une instruction `goto` *identificateur* a pour effet de transférer le contrôle d'exécution à l'instruction étiquetée par *identificateur*. L'instruction `goto` et l'instruction cible du `goto` doivent se trouver dans la même procédure : le langage C est un langage à branchement locaux.

- Exemple :

```
{
etiq2:
...            /*   des instructions                        */
goto etiq1;    /*   goto avant définition de l'étiquette    */
...            /*   des instructions                        */
etiq1:
...            /*   des instructions                        */
goto etiq2;    /*   goto après définition de l'étiquette    */
}
```

3.15.3 Instruction nulle

- Syntaxe :

  instruction :

  ⇒ ;

- Sémantique :

  ne rien faire !

Cette instruction ne rend que des services syntaxiques. Elle peut être utile quand on désire mettre une étiquette à la fin d'une instruction composée. Par exemple :

```
{
...
fin: ;
}
```

Elle est également utile pour mettre un corps nul à certaines instructions itératives. En effet, à cause des effets de bord dans les expressions, on peut arriver parfois à faire tout le travail dans les expressions de contrôle des instructions itératives. Par exemple, on peut initialiser à zéro un tableau de la manière suivante :

```
for (i = 0; i < N; t[i++]= 0)
    ;    /*   instruction nulle   */
```

Attention

Cette instruction nulle peut parfois avoir des effets désastreux. Supposons que l'on veuille écrire la boucle :

```
for (i = 0; i < N; i++)
    t[i] = i;
```

si par mégarde on met un `;` à la fin de ligne du `for`, on obtient un programme parfaitement correct, qui s'exécute sans broncher, mais ne fait absolument pas ce qui était prévu. En effet :

```
for (i = 0; i < N; i++) ;
    t[i] = i;
```

exécute le `for` avec le seul effet d'amener la variable `i` à la valeur N, et ensuite exécute une fois `t[i] = i` ce qui a probablement pour effet d'écraser la variable déclarée juste après le tableau `t`.

3.16 Exercice

Écrire une procédure `main` se comportant comme une calculette c'est à dire exécutant une boucle sur :

1. lecture d'une ligne supposée contenir un entier, un opérateur et un entier (ex : 12 + 34), les opérateurs seront `+ - * / %` ;

2. calculer la valeur de l'expression ;

3. imprimer le résultat.

Commentaire de la solution

Nous faisons ici un commentaire sur la solution proposée qui se trouve à la page suivante. Dans le cas où la ligne lue n'a pas une syntaxe correcte (elle ne contient pas un nombre, un signe, un nombre), le programme émet un message d'erreur et exécute `exit(1)`. Ceci ne réalise pas un interface utilisateur bien agréable, car il serait plus intéressant de continuer la boucle au lieu de terminer le programme. Cela n'a pas été implémenté car ce n'est pas réalisable à l'aide des seules possibilités de base de `scanf` qui ont été présentées. Dans le chapitre « Les entrées-sorties », `scanf` sera expliqué de manière exhaustive et une meilleure version de ce programme sera présentée.

```
#include <stdio.h>

enum {FAUX, VRAI};

/*****************************************************************************/
/*                                                                           */
/*                              main                                         */
/*                                                                           */
/*****************************************************************************/
int main()
{
int i,j,r;  /*   les opérandes  */
char c;     /*   l'opérateur   */
char imp;   /*   booléen de demande d'impression du résultat  */
int ret;    /*   code de retour de scanf   */

while (1)
   {
   if ((ret = scanf("%d %c %d",&i,&c,&j)) != 3)
      {
      if (ret == EOF) exit(0);
      printf("Erreur de syntaxe\n");
      exit(1);
      }
   imp = VRAI;
   switch (c)
      {
      case '+' : r = i + j; break;
      case '-' : r = i - j; break;
      case '*' : r = i * j; break;
      case '/' :
         if ( j == 0)
            {
            printf("Division par zéro\n");
            imp = FAUX;
            }
         else r = i / j;
         break;
      case '%' : r = i % j; break;
      default  : printf("l'opérateur %c est incorrect\n",c); imp = FAUX;
      }   /*   fin du switch   */

   if (imp) printf("%d\n",r);
   }
}
```

3.17 Récréation

Pour les amateurs de palindromes, voici la contribution de Brian Westley a la compétition du code C le plus obscur (IOCCC) de 1987.

```
                                char rahc
                                   [ ]
                                    =
                                  "\n/"
                                    ,
                                redivider
                                   [ ]
                                    =
                       "Able was I ere I saw elbA"
                                    ,
                                    *
                             deliver,reviled
                                    =
                                   1+1
                                    ,
                               niam ; main
                                   ( )
                                  {/*\}
                                   \*/
                                 int tni
                                    =
                                   0x0
                                    ,
                             rahctup,putchar
                                   ( )
                           ,LACEDx0 = 0xDECAL,
                                rof ; for
                             (;(int) (tni);)
                               (int) (tni)
                          = reviled ; deliver =
                                redivider
                                    ;
for ((int)(tni)++,++reviled;reviled* *deliver;deliver++,++(int)(tni)) rof
                                    =
                             (int) -1- (tni)
                          ;reviled--;--deliver;
                             (tni)  =  (int)
                          - 0xDECAL + LACEDx0 -
                                rof ; for
       (reviled--,(int)--(tni);(int) (tni);(int)--(tni),--deliver)
                            rahctup = putchar
                           (reviled* *deliver)
                                    ;
                            rahctup * putchar
                            ((char) * (rahc))
                                    ;
                                   /*\
                                  {\*/}
```

L'exécution de ce programme imprime le palindrome : « Able was I ere I saw elbA ».

Réponse de la devinette

Non, `*i/*j` n'est pas un quotient, car `/*` est un début de commentaire. Pour obtenir le quotient de `*i` par `*j` il faut utiliser du blanc : `*i / *j`.

Chapitre 4

Relations entre tableaux et pointeurs

4.1 Conversion des tableaux

Nous avons jusqu'à présent utilisé les tableaux de manière intuitive, en nous contentant de savoir qu'on peut déclarer un tableau par une déclaration du genre :

```
int t[10];
```

et qu'on dispose d'un opérateur d'indexation noté `[]`, permettant d'obtenir un élément du tableau : l'élément d'index `i` du tableau `t` se désigne par `t[i]`. Il est temps maintenant d'exposer une caractéristique originale des références à un tableau dans le langage C : elles subissent une conversion automatique.

Règle :

Tout identificateur de type « tableau de `X` » apparaissant dans une expression est converti en une valeur constante dont :

– le type est « pointeur vers `X` » ;

– la valeur est l'adresse du premier élément du tableau.

Cette conversion n'a lieu que pour un identificateur de type « tableau de `X` » **apparaissant dans une expression**. En particulier, elle n'a pas lieu lors de la déclaration. Quand on déclare `int T[10]`, le compilateur mémorise que `T` est de type « tableau de 10 int » et réserve de la place en mémoire pour 10 entiers. C'est lors de toute utilisation ultérieure de l'identificateur `T`, que cette occurrence de `T` sera convertie en type `int *`, de valeur adresse de `T[0]`.

Remarques

1. La conversion automatique d'un identificateur ayant le type tableau empêche de désigner un tableau en entier, c'est pour cette raison que l'opérateur d'affectation ne peut affecter un tableau à un autre tableau :

   ```
   int t1[10];
   ```

```
int t2[10];
t1 = t2;     /*   le compilateur rejettera cette instruction   */
```

Un telle affectation ne peut se réaliser qu'à l'aide d'une procédure qui réalisera l'affectation élément par élément.

2. Un identificateur ayant le type tableau est converti en une valeur constante, on ne peut donc rien lui affecter :

```
int *p;
int t[10];
t = p;      /*   interdit   */
p = t;      /*   valide     */
```

L'existence des conversions sur les références aux tableaux va avoir deux conséquences importantes : la première concerne l'opérateur d'indexation et la seconde le passage de tableaux en paramètre.

4.2 L'opérateur d'indexation

La sémantique de l'opérateur d'indexation consiste à dire qu'après les déclarations :

```
int t[N];
int i;
```

`t[i]` est équivalent à `*(t + i)`. Vérifions que cela est bien conforme à la façon dont nous l'avons utilisé jusqu'à présent. Nous avons vu que `t` à pour valeur l'adresse du premier élément du tableau. D'après ce que nous savons sur l'addition entre un pointeur et un entier, nous pouvons conclure que `t + i` est l'adresse de l'élément de rang `i` du tableau. Si nous appliquons l'opérateur d'indirection à `(t+i)` nous obtenons l'élément de rang `i` du tableau, ce que nous notions jusqu'à présent `t[i]`.

conséquence numéro 1

L'opérateur d'indexation noté `[]` est donc inutile, et n'a été offert que pour des raisons de lisibilité des programmes, et pour ne pas rompre avec les habitudes de programmation.

conséquence numéro 2

Puisque l'opérateur d'indexation s'applique à des valeurs de type pointeur, on va pouvoir l'appliquer à n'importe quelle valeur de type pointeur, et pas seulement aux constantes repérant des tableaux. En effet, après les déclarations :

```
int t[10];
int * p;
```

on peut écrire :

```
p = &t[4];
```

et utiliser l'opérateur d'indexation sur `p`, `p[0]` étant `t[4]`, `p[1]` étant `t[5]`, etc. `p` peut donc être utilisé comme un sous-tableau de `t`.

conséquence numéro 3

L'opérateur d'indexation est commutatif ! En effet, `t[i]` étant équivalent à `*(t + i)` et l'addition étant commutative, `t[i]` est équivalent à `*(i + t)` donc à `i[t]`. Lorsqu'on utilise l'opérateur d'indexation, on peut noter indifféremment l'élément de rang i d'un tableau, `t[i]` ou `i[t]`. Il est bien évident que pour des raisons de lisibilité, une telle notation doit être prohibée, et doit être considérée comme une conséquence pathologique de la définition de l'opérateur d'indexation.

4.3 Passage de tableau en paramètre

Du fait de la conversion d'un identificateur de type tableau en l'adresse du premier élément, lorsqu'un tableau est passé en paramètre effectif, c'est cette adresse qui est passée en paramètre. Le paramètre formel correspondant devra donc être déclaré comme étant de type pointeur.

Voyons sur un exemple. Soit à écrire une procédure `imp_tab` qui est chargée d'imprimer un tableau d'entiers qui lui est passé en paramètre. On peut procéder de la manière suivante :

```
void imp_tab(int *t, int nb_elem)   /*   définition de imp_tab       */
{
int i;

for (i = 0; i < nb_elem; i++) printf("%d ",*(t + i));
}
```

Cependant, cette méthode a un gros inconvénient. En effet, lorsqu'on lit l'en-tête de cette procédure, c'est à dire la ligne :

```
void imp_tab(int *t, int nb_elem)
```

il n'est pas possible de savoir si le programmeur a voulu passer en paramètre un pointeur vers un `int` (c'est à dire un pointeur vers **un seul** `int`), ou au contraire si il a voulu passer un tableau, c'est à dire un pointeur vers une zone de n `int`. De façon à ce que le programmeur puisse exprimer cette différence dans l'en-tête de la procédure, le langage C admet que l'on puisse déclarer un paramètre formel de la manière suivante :

```
void proc(int t[])
{
...   /*   corps de la procédure proc   */
}
```

car le langage assure que lorsqu'un paramètre formel de procédure ou de fonction est déclaré comme étant de type tableau de X, il est considéré comme étant de type pointeur vers X.

Si d'autre part, on se souvient que la notation `*(t + i)` est équivalente à la notation `t[i]`, la définition de `imp_tab` peut s'écrire :

```
void imp_tab(int t[], int nb_elem)   /*   définition de imp_tab       */
{
int i;

for (i = 0; i < nb_elem; i++) printf("%d ",t[i]);
}
```

Cette façon d'exprimer les choses est beaucoup plus claire, et sera donc préférée. L'appel se fera de la manière suivante :

```
#define NB_ELEM 10
int tab[NB_ELEM];

int main()
{
imp_tab(tab,NB_ELEM);
}
```

Remarque

Quand une fonction admet un paramètre de type tableau, il y à deux cas possibles :

- soit les différents tableaux qui lui sont passés en paramètre effectif ont des tailles différentes, et dans ce cas la taille doit être un paramètre supplémentaire de la fonction, comme dans l'exemple précédent ;
- soit les différents tableaux qui lui sont passés en paramètre effectif ont tous la même taille, et dans ce cas la taille peut apparaître dans le type du paramètre effectif :

  ```
  #define NB_ELEM 10
  void imp_tab(int t[NB_ELEM])
  {
  ...
  }
  ```

4.4 Modification des éléments d'un tableau passé en paramètre

Lorsqu'on passe un paramètre effectif à une procédure ou une fonction, on a vu que l'on passait une valeur. Il est donc impossible à une procédure de modifier la valeur d'une variable passée en paramètre.

En ce qui concerne les tableaux par contre, on passe à la procédure l'adresse du premier élément du tableau. La procédure pourra donc modifier si elle le désire les éléments du tableau.

Il semble donc que le passage de tableau en paramètre se fasse par adresse et non par valeur et qu'il s'agisse d'une exception à la règle qui affirme qu'en C, tout passage de paramètre se fait par valeur. Mais il n'en est rien : c'est la conversion automatique des identificateurs de type tableau qui provoque ce phénomène.

Du point de vue pratique, on retiendra que l'on peut modifier les éléments d'un tableau passé en paramètre. On peut écrire par exemple :

```
/*   incr_tab fait + 1 sur tous les éléments du tableau t  */
void incr_tab(int t[], int nb_elem)
{
int i;

for (i = 0; i < nb_elem; i++) t[i]++;
}
```

4.5 Interdiction de modification des éléments d'un tableau passé en paramètre

Lors de la normalisation du langage C, le comité en charge du travail a pensé qu'il était important d'introduire dans le langage un mécanisme permettant au programmeur d'exprimer l'idée : « cette procédure qui admet en paramètre ce tableau, ne doit pas en modifier les éléments ». Pour réaliser cela, un nouveau mot-clé a été introduit, le mot `const`, qui permet de déclarer des variables de la manière suivante :

```
const int i = 10;
```

qui déclare une variable de nom `i` dont il est interdit de modifier la valeur. L'intérêt de `const` se manifeste pour les paramètres de fonction. Reprenons l'exemple de la procédure `imp_tab`, pour exprimer le fait que cette procédure ne doit pas modifier les éléments du tableau `t`, on peut (et il est recommandé de) l'écrire de la façon suivante :

```
void imp_tab(const int t[], int nb_elem)   /*   définition de imp_tab       */
{
int i;

for (i = 0; i < nb_elem; i++) printf("%d ",t[i]);
}
```

4.6 Conversion des chaînes littérales

On rappelle que les chaînes littérales peuvent être utilisées lors de la déclaration avec initialisation d'un tableau de caractères, comme ceci :

```
char message[] = "Bonjour !!";
```

ou être utilisées dans une expression et être passées en paramètre de fonction par exemple, comme cela :

```
printf("Bonjour");
```

Règle

Lorsque les chaînes littérales apparaissent dans un autre contexte qu'une déclaration avec initialisation de tableau de caractères, elles subissent une conversion en pointeur vers `char`. Si une fonction a comme paramètre formel un tableau de caractères, on pourra lui passer en paramètre effectif aussi bien le nom d'un tableau de caractères qu'une chaîne littérale. Exemple :

```
char mess[] = "Bonjour";
void f( char t[])
{
...    /*   corps de la fonction f  */
}

f(mess);         /*   un appel possible de f         */
f("Hello");      /*   un autre appel possible de f   */
```

4.7 Retour sur `printf`

Nous avons vu au paragraphe 1.16 que la fonction `printf` admet comme premier paramètre une chaîne à imprimer comportant éventuellement des séquence d'échappement : `%d`, `%o` et `%x` pour imprimer un nombre respectivement en décimal, octal et hexadécimal et `%c` pour imprimer un caractère. Il existe aussi une séquence d'échappement pour imprimer les chaînes de caractères :`%s`. Exemple :

```
char mess1[] = "Hello";
char mess2[] = "Bonjour";
char *p;
if (...) p = mess1; else p = mess2;
printf("Message = %s\n",p);
```

4.8 Exercice

1. Déclarer et initialiser deux tableaux de caractères (ch1 et ch2).
2. Écrire une fonction (`lg_chaine1`) qui admette en paramètre un tableau de caractères se terminant par un *null*, et qui rende le nombre de caractères du tableau (*null* exclu).
3. Écrire une fonction (`lg_chaine2`) qui implémente le même interface que `lg_chaine1`, mais en donnant à son paramètre le type pointeur vers `char`.
4. La procédure `main` imprimera le nombre d'éléments de `ch1` et `ch2` par un appel à `lg_chaine1` et `lg_chaine2`.

```c
#include <stdio.h>

#define NULL_C '\0'
char ch1[] = "cette chaîne comporte 35 caractères";
char ch2[] = "et celle ci fait 30 caractères";

/*****************************************************************************/
/*                                                                           */
/*                              lg_chaine1                                   */
/*                                                                           */
/*   But:                                                                    */
/*      calcule la longueur d'une chaîne de caractères                       */
/*                                                                           */
/*   Interface:                                                              */
/*      ch : la chaîne de caractères                                         */
/*      valeur rendue : la longueur de ch                                    */
/*                                                                           */
/*****************************************************************************/
int lg_chaine1(const char ch[])
{
int i = 0;

while (ch[i] != NULL_C) i++;   /*   équivalent a while(ch[i]) i++;   */

return(i);
}

/*****************************************************************************/
/*                                                                           */
/*                              lg_chaine2                                   */
/*                                                                           */
/*   But:                                                                    */
/*      identique à celui de lg_chaine1                                      */
/*                                                                           */
/*****************************************************************************/
int lg_chaine2(const char *ch)
{
int i = 0;

while (*ch != NULL_C)
   {   i++; ch++;  }

return(i);
}

/*****************************************************************************/
/*                              main                                         */
/*****************************************************************************/
int main()
{
printf("la longeur de ch1 est %d\n",lg_chaine1(ch1));
printf("la longeur de ch2 est %d\n",lg_chaine2(ch2));
}
```

4.9 Tableaux multidimensionnels

4.9.1 Déclarations

En C, un tableau multidimensionnel est considéré comme étant un tableau dont les éléments sont eux mêmes des tableaux. Un tableau à deux dimensions se déclare donc de la manière suivante :

```
int t[10][20];
```

Les mêmes considérations que celles que nous avons développées sur les tableaux à une dimension s'appliquent, à savoir :

1. à la déclaration, le compilateur allouera une zone mémoire permettant de stocker de manière contiguë 10 tableaux de 20 entiers, soit 200 entiers ;
2. toute référence ultérieure à `t` sera convertie en l'adresse de sa première ligne, avec le type pointeur vers tableau de 20 `int`.

4.9.2 Accès aux éléments

L'accès à un élément du tableau se fera de préférence par l'expression `t[i][j]`.

4.9.3 Passage en paramètre

Lorsqu'on désire qu'un paramètre formel soit un tableau à deux dimensions, il faut le déclarer comme dans l'exemple suivant :

```
#define N 10

p(int t[][N])
{
...   /*  corps de p   */
}
```

On peut en effet omettre la taille de la première dimension, mais il est nécessaire d'indiquer la taille de la seconde dimension, car le compilateur en a besoin pour générer le code permettant d'accéder à un élément. En effet, si T est la taille des éléments de `t`, l'adresse de `t[i][j]` est : *adresse de* $t + (i \times N + j) \times T$. Le compilateur à besoin de connaître N, ce sera donc une constante. Par contre, la taille de la première dimension pourra être passée en paramètre, comme nous l'avons fait pour les tableaux à une seule dimension. Exemple :

```
#define P 10

void raz_mat(int t[][P], int n)
{
int i,j;

for (i = 0; i < n; i++)
   for (j = 0; j < P; j++)
      t[i][j] = 0;
}
```

raz_mat ne sera appliquable qu'à des tableaux dont la première dimension à une taille quelconque, mais dont la seconde dimension doit impérativement avoir P éléments.

4.10 Initialisation

Un tableau multidimensionnel peut être initialisé à l'aide d'une liste de listes d'expressions constantes. Exemple :

```
int t[4][5] = {
   { 0,1,2,3,4},
   { 10,11,12,13,14},
   { 20,21,22,23,24},
   { 30,31,32,33,34},
};
```

Un telle initialisation doit se faire avec des expressions constantes, c'est à dire délivrant une valeur connue à la compilation.

4.11 Exercice

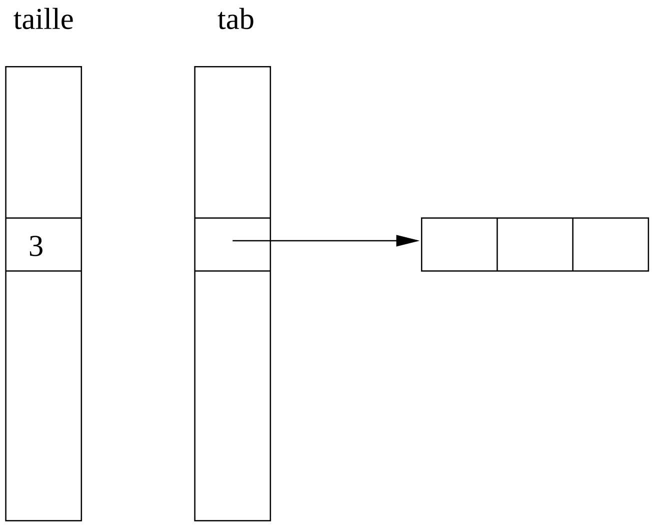
1. Déclarer et initialiser statiquement une matrice [5,5] d'entiers (tab).
2. Écrire une fonction (print_mat) qui admette en paramètre une matrice [5,5] et qui imprime ses éléments sous forme de tableau.
3. La procédure main fera un appel à print_mat pour le tableau tab.

```
#include <stdio.h>

#define N 5

int tab[N][N] =
   {
   {0,1,2,3,4},
   {10,11,12,13,14},
   {20,21,22,23,24},
   {30,31,32,33,34},
   {40,41,42,43,44}
   };

/*****************************************************************************/
/*                                                                           */
/*                              print_mat                                    */
/*                                                                           */
/*   But:                                                                    */
/*      Imprime un tableau N sur N (N est une constante)                     */
/*                                                                           */
/*****************************************************************************/
print_mat(int t[][N])
{
int i,j;

for (i= 0; i < N; i++)
   {
   for (j = 0; j < N; j++)
      printf("%d ",t[i][j]);
   printf("\n");
   }
}

/*****************************************************************************/
/*                                                                           */
/*                              main                                         */
/*                                                                           */
/*****************************************************************************/
int main()
{
print_mat(tab);
}
```

4.12 Tableau de pointeurs

4.12.1 Cas général

Pour des raisons de gain de place mémoire, on est parfois amené à créer des tableaux à deux dimensions dont toutes les lignes n'ont pas la même taille. Ceci peut se réaliser à l'aide d'un tableau de pointeurs vers des tableaux de tailles différentes, associé à un autre tableau qui donnera la taille de chaque ligne. On obtient alors la structure de données suivante :



Si nous supposons que le type des objets terminaux est `int`, pour traduire cette structure de données en langage C, les programmeurs ont l'habitude de « tricher », et de ne pas utiliser le type *tableau de pointeurs vers des tableaux de* `int`, considéré comme étant trop compliqué (un tel type s'écrit : `int (*tab[NB_ELEM])[]`). La solution habituellement retenue, consiste à utiliser le type *tableau de pointeurs vers des* `int`, soit `int *tab[NB_ELEM]`. Voici un exemple d'une telle déclaration avec initialisation statique du tableau :

```
#define NB_ELEM 3
int taille[NB_ELEM] = {1, 2, 3};
int ligne1[] = {10};
int ligne2[] = {20,21};
int ligne3[] = {30,31,32};

int *tab[] = {ligne1, ligne2, ligne3};
```

Pour une référence à l'élément j de la ligne i, on n'a que l'embarras du choix. Nous donnons ci-après trois méthodes différentes d'imprimer les éléments du tableau ligne par ligne.

Première méthode

On utilise un pointeur vers un entier que l'on fait progresser d'élément en élément dans une ligne :

```
int i, *p;
for (i = 0; i < NB_ELEM; i++)
   {
   for (p = tab[i]; p < tab[i] + taille[i]; p++)
      printf("%d ",*p);           /*   accès à l'élément courant par *p   */
   printf("\n");
   }
```

Deuxième méthode

On utilise un pointeur vers le premier élément d'une ligne ; ce pointeur reste fixe.

```
int i, j, *p;
for (i = 0; i < NB_ELEM; i++)
   {
   for (p = tab[i], j = 0; j < taille[i]; j++)
      printf("%d ",p[j]);         /*   accès à l'élément courant par p[j]   */
   printf("\n");
   }
```

Troisième méthode

La dernière méthode est surprenante : pour la comprendre, il suffit de remarquer que la variable `p` dans la seconde solution est inutile, on peut accéder à l'élément courant par la notation `tab[i][j]`.

```
int i, j, *p;
for (i = 0; i < NB_ELEM; i++)
   {
   for (j = 0; j < taille[i]; j++)
      printf("%d ", tab[i][j]);  /* accès à l'élément courant par tab[i][j] */
   printf("\n");
   }
```

On remarquera que c'est la même notation que celle qui est utilisée quand on a un vrai tableau à deux dimensions, c'est à dire une structure de données physiquement complètement différente. Que l'accès à deux structures de données différentes puissent se réaliser de la même manière, doit sans doute être considéré comme une faiblesse du langage.

4.12.2 Tableaux de pointeurs vers des chaînes

C'est pour les tableaux de caractères à deux dimensions, que se manifeste le plus souvent l'intérêt de disposer d'un tableau de lignes de longueurs différentes : les longueurs des chaînes sont extrêmement variables. La aussi, les habitudes sont les mêmes, les programmeurs utilisent le type *tableau de pointeurs vers des char*[1], comme ceci :

```
char * t[NB_ELEM];
```

1. Alors qu'en toute rigueur il faudrait utiliser le type tableau de pointeurs vers des tableaux de char

On peut initialiser un tableau de ce type avec des chaînes littérales :

```
char * mois[] = {"janvier", "février", "mars", "avril", "mai", "juin",
           "juillet", "août", "septembre", "octobre", "novembre", "décembre"};
```

On remarquera que ceci est impossible avec tout autre type que les `char` : il est impossible d'écrire :

```
int * tab[] = {{1}, {2,3}, {4,5,6}};
```

Une boucle d'impression des valeurs du tableau `mois` pourra être :

```
#define NBMOIS 12
int i;

for (i = 0; i < NBMOIS ; i++)
   printf("%s\n",mois[i]);
```

4.12.3 Paramètres d'un programme

Les tableaux de pointeurs vers des chaînes de caractères sont une structure de données importante, car c'est sur celle-ci que s'appuie la transmission de paramètres lors de l'exécution d'un programme. Lorsqu'un utilisateur lance l'exécution du programme `prog` avec les paramètres `param1`, `param2`, ... `paramn`, l'interpréteur de commandes collecte tous ces mots sous forme de chaînes de caractères, crée un tableau de pointeurs vers ces chaînes, et lance la procédure `main` en lui passant deux paramètres :

– un entier contenant la taille du tableau ;

– le tableau de pointeurs vers les chaînes.

Pour que le programme puisse exploiter les paramètres passés par l'utilisateur, la fonction `main` doit être déclarée de la manière suivante :

```
int main(int argc, char *argv[])
{
...
}
```

Les noms `argc` (pour *argument count*), ainsi que `argv` (pour *argument values*), sont des noms traditionnels, mais peuvent être remplacés par n'importe quels autres noms ; seuls les types doivent être respectés.

Comme exemple d'utilisation des paramètres, nous donnons le source d'un programme qui imprime son nom et ses paramètres :

```
int main(int argc, char *argv[])
{
int i;

printf("Nom du programme : %s\n", argv[0]);
for (i = 1; i < argc; i++)
   printf("Paramètre %d : %s\n",i,argv[i]);
}
```

4.13 Tableau et pointeur, c'est la même chose ?

Il y a un moment dans l'étude du langage C, où on a l'impression que les tableaux et les pointeurs sont plus ou moins interchangeables, en un mot que c'est pratiquement la même chose. il faut donc être très clair : un tableau et un pointeur **ce n'est pas** la même chose. Quand on déclare, ailleurs qu'en paramètre formel de fonction, `int t[10];` on déclare un tableau, et le compilateur réserve une zone de 10 entiers consécutifs. Quand on déclare `int *p`, il s'agit toujours d'un pointeur, et le compilateur réserve simplement un élément de mémoire pouvant contenir un pointeur.

Les caractéristiques du langage C qui créent la confusion dans l'esprit des utilisateurs, sont les trois règles suivantes :

1. tout identificateur de type tableau de x apparaissant dans une expression est converti en une valeur constante de type pointeur vers x, et ayant comme valeur l'adresse du premier élément du tableau ;
2. un paramètre formel de fonction, de type tableau de x est considéré comme étant de type pointeur vers x ;
3. la sémantique de l'opérateur d'indexation est la suivante : `T[i]` est équivalent à `*(T + i)`.

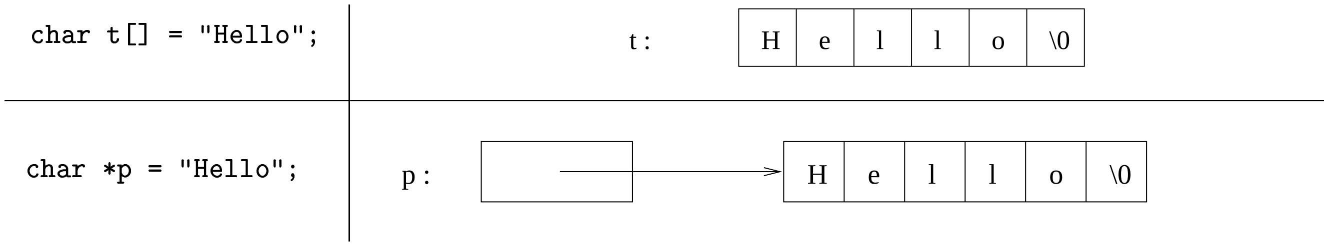
4.13.1 Commentaires

Bien noter les points suivants :

- Le point important à comprendre dans la règle 2 est que tableau de x est la même chose que pointeur vers x **uniquement dans le cas de paramètre formel de fonction**. Donc `void fonc (int t[]) { ... }` est équivalent à `void fonc (int * t) { ... }`. Les types des objets déclarés de type tableau ou de type pointeur sont différents dans tous les autres contextes, que ce soit déclaration de variables globales ou locales à une fonction.
- Différence entre règle 1 et règle 2 : une déclaration `int t[10]` qui n'est pas un paramètre formel, déclare un tableau de 10 entiers. Ce sont les utilisations ultérieures de `t` qui subissent une conversion en type pointeur vers entier. Par contre, à la déclaration d'un paramètre formel `int t[]`, c'est la déclaration elle-même qui est transformée en `int *t`.

4.13.2 Cas particulier des chaînes littérales

Les chaînes littérales viennent ajouter à la confusion, car on peut déclarer `char t[] = "Hello";` et `char *p = "Hello";`. Dans le premier cas, le compilateur alloue un tableau de 6 caractères qu'il initialise avec les caractères H, e, l, l, o et `\0`. Toute occurrence de `t` dans une expression sera convertie en type pointeur vers `char`. Dans le second cas, le compilateur alloue un tableau de 6 caractères qu'il initialise de la même manière que dans le premier cas, mais de surcroît, il alloue une variable de type pointeur vers `char` qu'il initialise avec l'adresse du premier caractère de la chaîne.



Ceci est un cas particulier des tableaux de caractères qui ne se reproduit pas avec les autres types. On peut déclarer un tableau d'entiers par `int t[] = {1, 2, 3, 4, 5};`, mais on ne peut pas déclarer un pointeur vers un tableau d'entiers par : `int *p = {1, 2, 3, 4, 5};`.

4.14 Récréation

En illustration sur les bizarreries des tableaux dans le langage C, voici la contribution de David Korn (le créateur du *korn shell*) à la compétition du code C le plus obscur (IOCCC) de 1987 :

```
main() { printf(&unix["\021%six\012\0"],(unix)["have"]+"fun"-0x60);}
```

Non, ce programme n'imprime pas *have fun with unix* ou quelque chose de ce genre ! Le lecteur est invité à essayer d'élucider ce programme (oui, il imprime quelque chose, mais quoi ?) la solution est donnée à la page suivante.

Voici les clés de la compréhension :

1. Tout compilateur C possède un certain nombre de constantes prédéfinies dépendant de l'architecture de la machine sur laquelle il s'exécute. En particulier, tout compilateur pour machine UNIX prédéfinit la constante `unix` avec comme valeur 1. Donc le programme est équivalent à :

   ```
   main() { printf(&1["\021%six\012\0"],(1)["have"]+"fun"-0x60);}
   ```

2. Si on se souvient qu'on a vu en 4.2 que pour un tableau `t`, `t[i]` est équivalent à `i[t]`, on voit que `1["\021%six\012\0"]` est équivalent à `"\021%six\012\0"[1]` et `(1)["have"]` à `"have"[1]`. Donc `&1["\021%six\012\0"]` est l'adresse du caractère `%` dans la chaîne `"\021%six\012\0"` et `"have"[1]` est le caractère `'a'`. On peut donc réécrire le programme :

   ```
   main() { printf("%six\012\0", 'a' + "fun" -0x60);}
   ```

3. La fin d'une chaîne est signalée par un caractère *null* (`\0`) et le compilateur en met un à la fin de chaque chaîne littérale. Celui qui est ici est donc inutile. D'autre part, il existe une notation plus parlante pour le caractère de code `\012` (c'est à dire *new line*), il s'agit de la notation `\n`. Le programme peut donc se réécrire :

   ```
   main() { printf("%six\n", 'a' + "fun" -0x60);}
   ```

4. Le caractère `'a'` a pour code ASCII 0x61, donc `'a' -0x60` est égal à 1. Réécrivons le programme :

   ```
   main() { printf("%six\n","fun" + 1); }
   ```

5. `"fun" + 1` est l'adresse du caractère u dans la chaîne `"fun"`, le programme devient donc :

   ```
   main() { printf("%six\n","un"); }
   ```

   il imprime donc `unix`.

Chapitre 5

Les entrées-sorties

À ce moment-ci de l'étude du langage, le lecteur éprouve sans doute le besoin de disposer de moyens d'entrées-sorties plus puissants que les quelques possibilités de `printf` et `scanf` que nous avons présentées. Nous allons donc consacrer un chapitre entier à cette question, en présentant les primitives les plus utiles de la bibliothèque standard. Toutes les primitives ne sont pas présentées, mais celles qui le sont, sont présentées de manière exhaustive et conforme à la norme ANSI.

5.1 Pointeur invalide

Quand on programme des listes chaînées, on a besoin de disposer d'une valeur de pointeur invalide pour indiquer la fin de la liste. Dans le fichier d'include `stddef.h` se trouve la définition d'une telle valeur qui porte le nom de `NULL`. Un certain nombre de fonctions de la bibliothèque standard qui doivent rendre un pointeur, utilisent la valeur `NULL` comme indicateur d'erreur.

5.2 Ouverture et fermeture de fichiers

5.2.1 Ouverture d'un fichier : `fopen`

Lorsqu'on désire accéder à un fichier, il est nécessaire avant tout accès d'ouvrir le fichier à l'aide de la fonction `fopen`.

Utilisation

`fopen (`*nom-de-fichier* `,` *mode*`)`

Sémantique des paramètres

– *nom-de-fichier* est de type pointeur vers `char`. La chaîne pointée est le nom du fichier auquel on veut accéder.

– *mode* est de type pointeur vers `char`. La chaîne pointée indique le mode d'ouverture, elle peut être l'une des chaîne suivantes :

`"r"`	ouverture d'un fichier texte en lecture.
`"w"`	ouverture d'un fichier texte en écriture.
`"a"`	ouverture d'un fichier texte en écriture à la fin.
`"rb"`	ouverture d'un fichier binaire en lecture.
`"wb"`	ouverture d'un fichier binaire en écriture.
`"ab"`	ouverture d'un fichier binaire en écriture à la fin.
`"r+"`	ouverture d'un fichier texte en lecture/écriture.
`"w+"`	ouverture d'un fichier texte en lecture/écriture.
`"a+"`	ouverture d'un fichier texte en lecture/écriture à la fin.
`"r+b"` ou `"rb+`	ouverture d'un fichier binaire en lecture/écriture.
`"w+b"` ou `"wb+`	ouverture d'un fichier binaire en lecture/écriture.
`"a+b"` ou `"ab+`	ouverture d'un fichier binaire en lecture/écriture à la fin.

Remarque

La fonction `fopen` a été normalisée en ayant présent à l'esprit que certains systèmes font la distinction entre fichiers binaires et fichiers textes. Cette distinction n'a pas cours dans le système UNIX.

Valeur rendue

La fonction `fopen` retourne une valeur de type pointeur vers `FILE`, où `FILE` est un type prédéfini dans le fichier `stdio.h`.

– Si l'ouverture a réussi, la valeur retournée permet de repérer le fichier, et devra être passée en paramètre à toutes les procédures d'entrées-sorties sur le fichier.

– Si l'ouverture s'est avérée impossible, `fopen` rend la valeur `NULL`.

Conditions particulières et cas d'erreur

– Si le mode contient la lettre `r`, le fichier doit exister, sinon c'est une erreur.

– Si le mode contient la lettre `w`, le fichier peut, ou peut ne pas, exister. Si le fichier n'existe pas, il est créé ; si le fichier existe déjà, son ancien contenu est perdu.

– Si le mode contient la lettre `a`, le fichier peut, ou peut ne pas, exister. Si le fichier n'existe pas, il est créé ; si le fichier existe déjà, son ancien contenu est conservé.

– Si un fichier est ouvert en mode « écriture à la fin », toutes les écritures se font à l'endroit qui est la fin du fichier au moment de l'exécution de l'ordre d'écriture. Cela signifie que si plusieurs processus partagent le même `FILE *`, résultat de l'ouverture d'un fichier en écriture à la fin, leurs écritures ne s'écraseront pas mutuellement. D'autre part, si un processus ouvre un fichier en écriture à la fin, fait un `fseek` pour se positionner à un endroit du fichier, puis fait une écriture, celle-ci aura lieu à la fin du fichier (pas nécessairement à l'endroit du `fseek`).

Les voies de communication standard

Quand un programme est lancé par le système, celui-ci ouvre trois fichiers correspondant aux trois voies de communication standard : *standard input*, *standard output* et *standard error*. Il y a trois constantes prédéfinies dans `stdio.h` de type pointeur vers `FILE` qui repèrent ces trois fichiers. Elles ont pour nom respectivement `stdin`, `stdout` et `stderr`.

Utilisation typique de `fopen`

```
#include <stdio.h>
FILE *fp;

if ((fp = fopen("donnees","r")) == NULL)
   {
   fprintf(stderr,"Impossible d'ouvrir le fichier données en lecture\n");
   exit(1);
   }
```

5.2.2 fermeture d'un fichier : `fclose`

Quand on a terminé les E/S sur un fichier, il faut en informer le système à l'aide de la fonction `fclose`.

Utilisation

`fclose` (*flot-de-données*)

Sémantique des paramètres

- *flot-de-données* est de type pointeur vers `FILE`. Il pointe vers le fichier à fermer.

Valeur rendue

La fonction `fclose` rend la valeur zéro si le fichier a été fermé, et rend `EOF` si il y a eu une erreur.

Utilisation typique

```
#include <stdio.h>
FILE *f;

fclose(f);
```

5.3 Lecture et écriture par caractère sur fichier

5.3.1 lecture par caractère: `fgetc`

Utilisation

`fgetc` (*flot-de-données*)

Sémantique des paramètres

- *flot-de-données* est de type pointeur vers `FILE`. Il pointe vers le fichier à partir duquel se fait la lecture.

Description

La fonction `fgetc` lit un caractère du fichier *flot-de-données*.

Valeur rendue

Si la lecture se fait sans erreur et sans rencontre de la fin de fichier, `fgetc` rend le caractère lu. Si il y a erreur d'entrée-sortie, ou rencontre de fin de fichier, `fgetc` rend la valeur `EOF`. Pour cette raison, le type de la valeur rendue est `int` et non pas `char`.

Utilisation typique

```
#include <stdio.h>
int c;
FILE *fi;

while ((c = fgetc(fi)) != EOF)
   {
   ...   /*   utilisation de c   */
   }
```

5.3.2 lecture par caractère: `getc`

Il existe une fonction `getc` qui est rigoureusement identique à `fgetc` (même interface, même sémantique), sauf que `getc` est implémenté comme une macro et non comme une vraie fonction C. La différence est que `getc` peut évaluer plusieurs fois son paramètre *flot-de-données*, ce qui lui interdit d'être une expression contenant des effets de bord. Exemple:

```
int i;
FILE * TAB_FILE[10];

c = getc(TAB_FILE[i++]);   /*   Arrgh..., effet de bord !   */
```

5.3.3 lecture par caractère : `getchar`

Utilisation

`getchar ( )`

Description

La fonction `getchar` est rigoureusement équivalente à `getc(stdin)`. C'est également une macro, mais comme elle n'admet pas de paramètre, elle n'a pas le (petit) inconvénient de `getc`.

5.3.4 écriture par caractère : `fputc`

Utilisation

`fputc (`*carac* `,` *flot-de-données*`)`

Sémantique des paramètres

- *carac* est de type `int`, c'est le caractère à écrire.
- *flot-de-données* est de type pointeur vers `FILE`. Il pointe vers le fichier sur lequel se fait l'écriture.

Description

La fonction `fputc` écrit le caractère *carac* dans le fichier *flot-de-données*.

Valeur rendue

La fonction `fputc` rend le caractère écrit si l'écriture s'est faite sans erreur, ou `EOF` en cas d'erreur.

Utilisation typique

```
#include <stdio.h>
int c;
FILE *fi,*fo;

/*   fi : résultat de fopen de fichier en lecture   */
/*   fo : résultat de fopen de fichier en écriture  */

while ((c = fgetc(fi)) != EOF)
   fputc(c,fo);
```

5.3.5 écriture par caractère : `putc`

Il existe une fonction `putc` qui est rigoureusement identique à `fputc` (même interface, même sémantique), sauf que `putc` est implémenté comme une macro et non comme une vraie fonction C. La même remarque que celle faite au sujet de `getc` s'applique donc à `putc`.

5.3.6 écriture par caractère : `putchar`

Utilisation

> `putchar (`*carac*`)`

Description

Un appel `putchar(c)` est rigoureusement identique à `fputc(c,stdout)`.

5.4 Lecture et écriture par lignes sur fichier

5.4.1 lecture par ligne : `fgets`

Utilisation

> `fgets (`*chaîne* , *taille* , *flot-de-données*`)`

Sémantique des paramètres

- *chaîne* est de type pointeur vers `char` et doit pointer vers un tableau de caractères.
- *taille* est la taille en octets du tableau de caractères pointé par *chaîne*.
- *flot-de-données* est de type pointeur vers `FILE`. Il pointe vers le fichier à partir duquel se fait la lecture.

Valeur rendue

La fonction `fgets` rend le pointeur *chaîne* cas de lecture sans erreur, ou `NULL` dans le cas de fin de fichier ou d'erreur.

Attention

> Sur fin de fichier ou erreur, `fgets` rend `NULL` et non pas `EOF`.
> Grrr ...

Description

La fonction `fgets` lit les caractères du fichier et les range dans le tableau pointé par *chaîne* jusqu'à rencontre d'un *line-feed* (qui est mis dans le tableau), ou rencontre de fin de fichier, ou jusqu'à ce qu'il ne reste plus qu'un seul caractère libre dans le tableau. `fgets` complète alors les caractères lus par un caractère *null*.

Utilisation typique

```
#include <stdio.h>
#define LONG ...
char ligne[LONG];
FILE *fi;

while (fgets(ligne,LONG,fi) != NULL)   /*  stop sur fin de fichier ou erreur  */
   {
   ...   /*   utilisation de ligne   */
   }
```

5.4.2 lecture par ligne : `gets`

Utilisation

`gets` (*chaîne*)

Sémantique des paramètres

- *chaîne* est de type pointeur vers `char` et doit pointer vers un tableau de caractères.

Valeur rendue

`gets` rend un pointeur vers le tableau de caractères en cas de lecture sans erreur, ou `NULL` dans le cas de fin de fichier ou d'erreur.

Description

La fonction `gets` est un `fgets` sur `stdin` avec la différence que le *line feed* n'est pas mis dans *chaîne*. Malheureusement, l'interface de `gets` est une catastrophe : il n'a pas le paramètre *taille* qui donne la taille du tableau pointé par *chaîne*. Ceci interdit donc à `gets` toute vérification pour ne pas déborder du tableau.
Pour cette raison l'usage de `gets` est très fortement déconseillé[1]

5.4.3 écriture par chaîne : `fputs`

Utilisation

`fputs` (*chaîne* , *flot-de-données*)

Sémantique des paramètres

- *chaîne* est de type pointeur vers `char`. Pointe vers un tableau de caractères contenant une chaîne se terminant par un *null*.

1. L'attaque d'Internet connue sous le nom de « the Internet worm » a profité de la présence dans le système d'un démon qui utilisait `gets`.

– *flot-de-données* est de type pointeur vers `FILE`. Il pointe vers le fichier sur lequel se fait l'écriture.

Valeur rendue

La fonction `fputs` rend une valeur non négative si l'écriture se passe sans erreur, et `EOF` en cas d'erreur.

Description

`fputs` écrit sur le fichier le contenu du tableau dont la fin est indiquée par un caractère *null*. Le tableau de caractères peut contenir ou non un *line-feed*. `fputs` peut donc servir indifféremment à écrire une ligne ou une chaîne quelconque.

Utilisation typique

```
#include <stdio.h>
#define LONG ...
char ligne[LONG];
FILE *fo;

fputs(ligne,fo);
```

5.4.4 écriture par chaîne : `puts`

Utilisation

`puts (`*chaîne*`)`

Sémantique des paramètres

– *chaîne* est de type pointeur vers `char`. Pointe vers un tableau de caractères contenant une chaîne se terminant par un *null*.

Valeur rendue

La fonction `fputs` rend une valeur non négative si l'écriture se passe sans erreur, et `EOF` en cas d'erreur.

Description

La fonction `puts` est un `fputs` sur `stdout`. Elle n'est pas entaché du même vice rédhibitoire que `gets`, on peut donc l'utiliser.

5.5 E/S formattées sur fichiers

5.5.1 Écriture formattée : `fprintf`

Utilisation

La fonction `fprintf` admet un nombre variable de paramètres. Son utilisation est la suivante :
`fprintf (` *flot-de-données* `,` *format* `,` $param_1$ `,` $param_2$ `,` ... `,` $param_n$ `)`

Sémantique des paramètres

- *flot-de-données* est de type pointeur vers `FILE`. Il pointe vers le fichier sur lequel se fait l'écriture.

- *format* est une chaîne de caractères qui spécifie ce qui doit être écrit.

- $param_i$ est une expression délivrant une valeur à écrire.

Valeur rendue

La fonction `fprintf` retourne le nombre de caractères écrits, ou une valeur négative si il y a eu une erreur d'entrée-sortie.

Présentation

La chaîne *format* contient des caractères ordinaires (c'est à dire différents du caractère `%`) qui doivent être copiés tels quels, et des séquences d'échappement (introduites par le caractère `%`), décrivant la manière dont doivent être écrits les paramètres $param_1$, $param_2$, ... $param_n$.

Si il y a moins de $param_i$ que n'en réclame le *format*, le comportement n'est pas défini. Si il y a davantage de $param_i$ que n'en nécessite le *format*, les $param_i$ en excès sont évalués, mais leur valeur est ignorée.

Les séquences d'échappement

Une séquence d'échappement se compose des éléments suivants :

1 Un certain nombre (éventuellement zéro) d'indicateurs pouvant être les caractères suivants :

`-`	$param_i$ sera cadré à gauche dans son champ d'impression.
`+`	si $param_i$ est un nombre signé, il sera imprimé précédé du signe `+` ou `-`.
blanc	si $param_i$ est un nombre signé et si son premier caractère n'est pas un signe, on imprimera un blanc devant $param_i$. Si on a à la fois l'indicateur `+` et l'indicateur *blanc*, ce dernier sera ignoré.

`#` cet indicateur demande l'impression de $param_i$ sous une forme non standard. Pour le format `o`, cet indicateur force la précision à être augmentée de manière à ce que $param_i$ soit imprimé en commençant par un zéro. Pour les formats `x` et `X`, cet indicateur a pour but de faire précéder $param_i$ respectivement de `0x` ou `0X`, sauf si $param_i$ est nul. Pour les formats `e`, `E`, `f`, `g` et `G`, cet indicateur force $param_i$ à être imprimé avec un point décimal, même si il n'y a aucun chiffre après. Pour les formats `g` et `G`, cet indicateur empêche les zéros de la fin d'être enlevés. Pour les autres formats, le comportement n'est pas défini.

`0` pour les formats `d`, `i`, `o`, `u`, `x`, `X`, `e`, `E`, `f`, `g`, et `G` cet indicateur a pour effet de compléter l'écriture de $param_i$ avec des `0` en tête, de manière à ce qu'il remplisse tout son champ d'impression. Si il y a à la fois les deux indicateurs `0` et `-`, l'indicateur `0` sera ignoré. Pour les formats `d`, `i`, `o`, `u`, `x` et `X`, si il y a une indication de précision, l'indicateur `0` est ignoré. Pour les autres formats, le comportement n'est pas défini.

2 Un nombre entier décimal (optionnel) indiquant la taille minimum du champ d'impression, exprimée en caractères. Si $param_i$ s'écrit sur un nombre de caractères inférieur à cette taille, $param_i$ est complété à droite ou à gauche (selon que l'on aura utilisé ou pas l'indicateur `-`), avec des blancs ou des `0`, comme il a été expliqué plus haut.

3 Une indication (optionnelle) de précision, qui donne :

- le nombre minimum de chiffres pour les formats `d`, `i`, `o`, `u`, `x` et `X`.
- le nombre de chiffres après le point décimal pour les formats `e`, `E` et `f`.
- le nombre maximum de chiffres significatifs pour les formats `g` et `G`
- le nombre maximum de caractères pour le format `s`.

Cette indication de précision prend la forme d'un point (`.`) suivi d'un nombre décimal optionnel. Si ce nombre est omis, la précision est prise égale à 0.

Remarque

Le nombre entier décimal indiquant la taille maximum du champ d'impression et/ou le nombre entier décimal indiquant la précision, peuvent être remplacées par le caractère `*`. Si le caractère `*` a été utilisé une seule fois dans le format, la valeur correspondante (taille du champ ou précision) sera prise égale à $param_{i-1}$. Si le caractère `*` a été utilisé deux fois, la taille du champ d'impression sera égale à $param_{i-2}$, et la précision sera égale à $param_{i-1}$. Si la taille maximum du champ d'impression a une valeur négative, cela a la sémantique de l'indicateur `-` suivi de la valeur (positive) de la taille du champ d'impression. Si la précision est négative, cela a la sémantique d'une précision absente.

4 un caractère optionnel, qui peut être :

- `h` s'appliquant aux formats `d`, `i`, `o`, `u`, `x` ou `X` : $param_i$ sera interprété comme un `short int` ou `unsigned short int`.
- `h` s'appliquant au format `n` : $param_i$ sera interprété comme un pointeur vers un `short int`.
- `l` [2]. s'appliquant aux formats `d`, `i`, `o`, `u`, `x` ou `X` : $param_i$ sera interprété comme un `long int` ou un `unsigned long int`.
- `l` [3] s'appliquant au format `n` : $param_i$ sera interprété comme un pointeur vers un `long int`.
- `L` s'appliquant aux formats `e`, `E`, `f`, `g` ou `G` : $param_i$ sera interprété comme un `long double`.

Si un `h`, `l` ou `L` s'applique à un autre format que ceux indiqués ci-dessus, le comportement est indéterminé.

5 un caractère qui peut prendre les valeurs suivantes :

- `d`, `i` $param_i$ sera interprété comme un `int`, et écrit sous forme décimale signée.
- `u` $param_i$ sera interprété comme un `unsigned int`, et écrit sous forme décimale non signée.
- `o` $param_i$ sera interprété comme un `int`, et écrit sous forme octale non signée.
- `x`, `X` $param_i$ sera interprété comme un `int`, et écrit sous forme hexadécimale non signée. La notation hexadécimale utilisera les lettres `abcdef` dans le cas du format `x`, et les lettres `ABCDEF` dans le cas du format `X`.

2. la lettre elle, pas le chiffre un
3. même remarque

Dans les cas qui précèdent, la précision indique le nombre minimum de chiffres avec lesquels écrire $param_i$. Si $param_i$ s'écrit avec moins de chiffres, il sera complété avec des zéros. La précision par défaut est 1. Une valeur nulle demandée avec une précision nulle, ne sera pas imprimée.

`c` — $param_i$ sera interprété comme un `unsigned char`.

`s` — $param_i$ sera interprété comme l'adresse d'un tableau de caractères (terminé ou non par un *null*). Les caractères du tableau seront imprimés jusqu'au premier des deux évènements possibles :

- impression de *précision* caractères de $param_i$.
- rencontre de *null* dans $param_i$.

Dans le cas où $param_i$ n'est pas terminé par un *null*, le format d'impression doit comporter une indication de *précision*.

`p` — $param_i$ sera interprété comme un pointeur vers `void`. Le pointeur sera imprimé sous une forme dépendante de l'implémentation.

`n` — $param_i$ sera interprété comme un pointeur vers un `int` auquel sera affecté le nombre de caractères écrits jusqu'alors par cette invocation de `fprintf`.

`e`,`E` — $param_i$ sera interprété comme un `double` et écrit sous la forme :
`-`$_{option}$ *pe* `.` *pf* `e` *signe exposant*
dans laquelle *pe* et *pf* sont respectivement partie entière et partie fractionnaire de la mantisse. La partie entière est exprimée avec un seul chiffre, la partie fractionnaire est exprimée avec un nombre de chiffres égal à la précision. La précision est prise égale à 6 par défaut. Si la précision est 0, le point décimal n'apparaît pas. L'exposant contient toujours au moins deux chiffres. Si $param_i$ est nul, l'exposant sera nul. Dans le cas du format `E`, la lettre `E` est imprimée à la place de `e`.

`f` — $param_i$ sera interprété comme un `double` et écrit sous la forme :
`-`$_{option}$ *pe* `.` *pf*
dans laquelle *pe* et *pf* sont respectivement partie entière et partie fractionnaire de la mantisse. La partie fractionnaire est exprimée avec un nombre de chiffres égal à la précision. La précision est prise égale à 6 par défaut. Si la précision est 0, le point décimal n'apparaît pas.

`g`,`G` — $param_i$ sera interprété comme un `double` et écrit sous le format `f`, ou le format `e`, selon sa valeur. Si $param_i$ a un exposant inférieur à -4 ou plus grand ou égal à la précision, il sera imprimé sous le format `e`, sinon il sera imprimé sous le format `f`. D'éventuels zéros à la fin de la partie fractionnaire sont enlevés. Le point décimal n'apparaît que si il est suivi d'au moins un chiffre. Dans ce qui précède, l'utilisation du format `G` implique l'utilisation du format `E` à la place du format `e`.

`%` — Un caractère `%` est écrit.

5.5.2 Écriture formattée : `printf`

Nous avons déjà vu `printf`, mais nous allons la définir ici formellement.

Utilisation

La fonction `printf` admet un nombre variable de paramètres. Son utilisation est la suivante :

`printf (` *format* , $param_1$, $param_2$, ... , $param_n$ `)`

Description

Un appel `printf(fmt, ...)` est rigoureusement identique à `fprintf(stdout,fmt,...)`.

5.5.3 Écriture formattée dans une chaîne : `sprintf`

Utilisation

La fonction `sprintf` admet un nombre variable de paramètres. Son utilisation est la suivante :

`sprintf (` *chaîne* , *format* , $param_1$, $param_2$, ... , $param_n$ `)`

Description

La fonction `sprintf` réalise le même traitement que la fonction `fprintf`, avec la différence que les caractères émis par `sprintf` ne sont pas écrits dans un fichier, mais dans le tableau de caractères *chaîne*. Un *null* est écrit dans *chaîne* en fin de traitement.

5.5.4 Exemples d'utilisation des formats

```
            source C                              resultat

printf("|%d|\n",1234);                            |1234|
printf("|%d|\n",-1234);                           |-1234|
printf("|%+d|\n",1234);                           |+1234|
printf("|%+d|\n",-1234);                          |-1234|
printf("|% d|\n",1234);                           | 1234|
printf("|% d|\n",-1234);                          |-1234|
printf("|%x|\n",0x56ab);                          |56ab|
printf("|%X|\n",0x56ab);                          |56AB|
printf("|%#x|\n",0x56ab);                         |0x56ab|
printf("|%#X|\n",0x56ab);                         |0X56AB|
printf("|%o|\n",1234);                            |2322|
printf("|%#o|\n",1234);                           |02322|

printf("|%10d|\n",1234);                          |      1234|
printf("|%10.6d|\n",1234);                        |    001234|
printf("|%.6d|\n",1234);                          |001234|
printf("|%*.6d|\n",10,1234);                      |    001234|
printf("|%*.*d|\n",10,6,1234);                    |    001234|

printf("|%f|\n",1.234567890123456789e5);          |123456.789012|
printf("|%.4f|\n",1.234567890123456789e5);        |123456.7890|
printf("|%.20f|\n",1.234567890123456789e5);       |123456.78901234567456413060|
printf("|%20.4f|\n",1.234567890123456789e5);      |         123456.7890|

printf("|%e|\n",1.234567890123456789e5);          |1.234568e+05|
printf("|%.4e|\n",1.234567890123456789e5);        |1.2346e+05|
printf("|%.20e|\n",1.234567890123456789e5);       |1.23456789012345674564e+05|
printf("|%20.4e|\n",1.234567890123456789e5);      |          1.2346e+05|

printf("|%.4g|\n",1.234567890123456789e-5);       |1.235e-05|
printf("|%.4g|\n",1.234567890123456789e5);        |1.235e+05|
printf("|%.4g|\n",1.234567890123456789e-3);       |0.001235|
printf("|%.8g|\n",1.234567890123456789e5);        |123456.79|
```

5.5.5 Entrées formattées : `fscanf`

Utilisation

La fonction **`fscanf`** admet un nombre variable de paramètres. Son utilisation est la suivante :

`fscanf` (*flot-de-données* , *format* , *param*$_1$, *param*$_2$, ... , *param*$_n$)

Sémantique des paramètres

- *flot-de-données* est de type pointeur vers `FILE`. Il pointe vers le fichier à partir duquel se fait la lecture.

– *format* est une chaîne de caractères qui spécifie la forme de l'entrée admissible dans *flot-de-données*.

– les $param_i$ sont des pointeurs. Ils pointent des variables dans lesquelles `fscanf` dépose les valeurs lues dans *flot-de-données*, après les avoir converties en binaire.

Valeur rendue

Si au moins un $param_i$ s'est vu affecter une valeur, `fscanf` retourne le nombre de $param_i$ affectés. Si il y a eu rencontre de fin de fichier ou erreur d'entrée-sortie avant toute affectation à un $param_i$, `fscanf` retourne `EOF`.

Description

`fscanf` lit une suite de caractères du fichier défini par *flot-de-données* en vérifiant que cette suite est conforme à la description qui en est faite dans *format*. Cette vérification s'accompagne d'un effet de bord qui consiste à affecter des valeurs aux variables pointées par les différents $param_i$.

Quelques définitions

flot d'entrée il s'agit de la suite de caractères lus du fichier défini par *flot-de-données*.

caractères blancs il s'agit des six caractères suivants : *espace*, *tab*, *line feed*, *new line*, *vertical tab* et *form feed*.

modèle un modèle est la description d'un ensemble de chaînes de caractères. Exemple : `%d` est le modèle des chaînes formées de chiffres décimaux, éventuellement signées.

conforme on dira qu'une chaîne est conforme à un modèle quand elle appartient à l'ensemble des chaînes décrites par le modèle. Exemple : `123` est conforme au modèle `%d`.

directive une directive peut être :

– une suite de caractères blancs qui est un modèle d'un nombre quelconque de caractères blancs. Exemple : un espace est un modèle pour un nombre quelconque d'espaces, ou d'un nombre quelconque d'espace et de *tab* mélangés, ou d'un nombre quelconque d'espaces, de *tab* et de *line-feed* mélangés etc.

– une suite de caractères ordinaires (c'est à dire qui ne sont ni des caractères blancs, ni le caractère `%`) qui est un modèle pour elle-même. Exemple : la chaîne `hello` est un modèle de la seule chaîne `hello`.

– des séquences d'échappement introduites par le caractère `%`. Ces séquences jouent un double rôle : elle sont à la fois un modèle des chaînes acceptables dans le flot d'entrée, et elles sont également des ordres de conversion de la chaîne lue et d'affectation du résultat à une variable pointée par le $param_i$ correspondant. Exemple : la directive `%d` est un modèle des nombres décimaux et un ordre de conversion de la chaîne lue en valeur binaire et d'affectation à l'entier pointé par le $param_i$ correspondant.

Les séquences d'échappement

Les séquences d'échappement se composent des éléments suivants :

1 le caractère `*` (optionnel) qui indique que la directive doit être considérée comme un pur modèle : pas de conversion de la chaîne lue, et pas d'affectation à un $param_i$.

2 un nombre (optionnel) qui indique la longueur maximum de la chaîne acceptable du flot d'entrée.

3 un caractère optionnel, qui est un modificateur de type, pouvant prendre l'une des valeurs suivantes :

`h`	s'appliquant aux formats `d`, `i`, `n` : $param_i$ sera interprété comme un pointeur vers un `short int`.
`h`	s'appliquant aux formats `o`, `u`, `x` : $param_i$ sera interprété comme un pointeur vers un `unsigned short int`.
`l`	s'appliquant aux formats `d`, `i`, `n` : $param_i$ sera interprété comme un pointeur vers un `long int`.
`l`	s'appliquant aux formats `o`, `u`, `x` : $param_i$ sera interprété comme un pointeur vers un `unsigned long int`.
`l`	s'appliquant aux formats `e`, `f`, `g` : $param_i$ sera interprété comme un pointeur vers un `double`.
`L`	s'appliquant aux formats `e`, `f`, `g` : $param_i$ sera interprété comme un pointeur vers un `long double`.

4 un caractère de conversion qui peut prendre l'une des valeurs suivantes :

`d` modèle pour un nombre décimal éventuellement précédé d'un signe. Le $param_i$ correspondant est interprété comme un pointeur vers un `int`, sauf si la directive contient un modificateur de type.

`i` modèle pour un nombre éventuellement précédé d'un signe, et ayant l'une des trois formes suivantes :

- un nombre décimal.
- un nombre débutant par `0` qui sera pris comme nombre octal.
- un nombre débutant par `0x` ou `0X`, qui sera pris comme nombre hexadécimal.

Le $param_i$ correspondant est interprété comme un pointeur vers un `int`, sauf si la directive contient un modificateur de type.

`o` modèle pour un nombre octal éventuellement précédé d'un signe. Le $param_i$ correspondant est interprété comme un pointeur vers un `unsigned int`, sauf si la directive contient un modificateur de type.

`u` modèle pour un nombre décimal éventuellement précédé d'un signe.[4] Le $param_i$ correspondant est interprété comme un pointeur vers un `unsigned int`, sauf si la directive contient un modificateur de type.

4. Non, il n'y a pas d'erreur : bien que ce format soit pour des `unsigned`, dans le flot d'entrée, le nombre peut être précédé d'un signe, et pas seulement `+`, d'ailleurs !

`x` modèle pour un nombre hexadécimal éventuellement précédé d'un signe. Le $param_i$ correspondant est interprété comme un pointeur vers un `unsigned int`, sauf si la directive contient un modificateur de type.

Remarque

> En ce qui concerne les formats `o`, `u`, et `x`, le lecteur a sans doute été surpris de voir que le $param_i$ correspondant est interprété comme un pointeur vers un `unsigned int` alors que la chaîne dans le flot d'entrée qui correspond à ces formats est un nombre éventuellement précédé d'un signe. Il n'y a pas d'erreur, c'est bien ce que dit la norme.

`c` modèle pour une suite de caractères dont le nombre est donné par *le nombre (optionnel) qui indique la longueur maximum de la chaîne acceptable du flot d'entrée* (Cf plus haut). Si ce nombre optionnel ne figure pas dans la directive, il est pris égal à 1. Le $param_i$ correspondant est interprété comme étant un pointeur vers un tableau de caractères suffisamment grand pour contenir la chaîne lue. Il n'y a pas de *null* rajouté à la fin de la chaîne lue dans le flot d'entrée.

`s` modèle pour une suite de caractères non blancs. Le $param_i$ correspondant est interprété comme un pointeur vers un tableau de caractères suffisamment grand pour contenir la chaîne lue plus un *null* terminal.

`e,f,g` modèle pour un flottant écrit selon la syntaxe d'une constante flottante du langage C. Le $param_i$ correspondant est interprété comme un pointeur vers un `float` , sauf si la directive contient un modificateur de type.

`[` Dans la chaîne *format*, ce caractère introduit une séquence particulière destinée à définir un *scanset*. La séquence est formée du caractère `[`, suivi d'une suite de caractères quelconques, suivi du caractère `]`. Si le premier caractère après le crochet ouvrant n'est pas le caractère `^`, le *scanset* est l'ensemble des caractères entre crochets. Si le caractère après le crochet ouvrant est le caractère `^`, le *scanset* est l'ensemble des caractères **ne** se trouvant **pas** dans la chaîne entre crochets. Le *scanset* peut comprendre le caractère `]` à condition de le mettre en début soit `[] ...]` ou `[^]...]` selon que l'on utilise la forme sans ou avec `^`. Le *scanset* peut contenir `^` à condition de ne pas le mettre en tête : `[...^...]`.

Une directive `[` est un modèle pour une suite de caractères appartenant au *scanset*. Le $param_i$ correspondant est interprété comme un pointeur vers un tableau de caractères suffisamment grand pour contenir la chaîne lue plus un *null* terminal.

`p` modèle pour un pointeur écrit d'une manière dépendant de l'implémentation, mais identique à l'impression par `printf` d'un pointeur selon le format `%p`. Le $param_i$ correspondant est interprété comme un pointeur vers un pointeur vers `void`.

`n` cette directive n'est pas un modèle. Elle ne sert qu'à mettre une valeur dans l'objet pointé par le $param_i$ correspondant. Le $param_i$

	correspondant est interprété comme un pointeur vers un `int` dans lequel `fscanf` écrit le nombre de caractères lus jusqu'à ce moment, dans le flot de données, par cette invocation de `fscanf`. L'exécution d'une directive `%n` n'augmente pas le nombre des $param_i$ affectés qui sera retourné par `fscanf` (Cf 5.5.5).
`%`	est un modèle pour le caractère `%`. La directive complète est `%%`.

Algorithme de `fscanf`

La chaîne *format* doit se composer d'un ensemble de directives. Il doit y avoir autant de $param_i$ que de directives demandant l'affectation d'une valeur. Si il n'y a pas suffisamment de $param_i$ pour le *format*, le comportement n'est pas défini. Si il y a davantage de $param_i$ que demandé par le *format*, les $param_i$ en excès sont évalués mais ils sont inutilisés.

La fonction `fscanf` exécute dans l'ordre chaque directive du *format*. Si une directive échoue, la fonction `fscanf` retourne à l'appelant.

- L'exécution d'une directive formée de caractères blancs, consiste à consommer dans le flot d'entrée la plus longue séquence possible de caractères blancs. Même si cette séquence est de taille nulle, la directive a réussi.

- L'exécution d'une directive formée de caractères ordinaires, consiste à consommer dans le flot d'entrée une séquence identique à la directive. Au premier caractère différent, la directive a échoué et ce caractère reste non lu.

- L'exécution d'une directive formée d'une séquence d'échappement, consiste à :

  1. consommer dans le flot d'entrée la plus longue séquence possible de caractères blancs. Cette séquence peut être de taille nulle. Cette action ne s'applique pas aux formats `c`, `n`, ni `[`.
  2. consommer dans le flot d'entrée la plus longue séquence possible de caractères qui soit conforme au modèle. Si cette séquence est de taille nulle, la directive a échoué.
  3. si la directive ne contient pas le caractère `*`, convertir la chaîne lue et l'affecter à l'objet pointé par le $param_i$ correspondant. Si cet objet n'est pas de la taille ou du type convenable pour la recevoir, le comportement n'est pas défini.

Remarques sur la gestion des *espaces blancs*

La gestion des espaces blancs est assez pénible. Il y a deux méthodes de consommation des espaces blancs du flot de données :

1. le *format* contient une directive formée de caractères blancs. Si une telle directive ne peut consommer aucun caractère blanc, c'est un cas d'échec de la directive.

2. le *format* contient une séquence d'échappement (autre que `c`, `n`, ou `[`) dont l'exécution commence par consommer d'éventuels caractères blancs dans le flot de données. Si il n'y a pas de caractères blancs à consommer, ce n'est pas une condition d'échec de la directive.

Voyons quelques conséquences.

Si le flot de données contient `23 45` on pourra les lire indifféremment soit :

- par le format `"%d %d"` : la première directive `%d` consomme `23`, la directive *blanc* (entre les deux `%d`) consomme les blancs entre `2` et `45`, et la seconde directive `%d` essaye de consommer des blancs, échoue (mais ce n'est pas une erreur), puis consomme `45`.
- par le format `"%d%d"` : la première directive `%d` consomme `23`, la seconde directive `%d` essaye de consommer des blancs, réussit, puis consomme `45`.

Dans les deux cas, les valeurs affectées seront bien les mêmes : 23 et 45.

Un tel phénomène ne se manifeste pas avec les séquences d'échappement dont l'exécution ne commence pas par consommer les espaces blancs. Par exemple, si le flot de données contient `23    jean dupond`, on n'obtiendra pas le même résultat selon que l'on utilise le format
`"%d %[ abcdefghijklmnopqrstuvwxyz]"` ou le format
`"%d%[ abcdefghijklmnopqrstuvwxyz]"` (sans blanc après `%d`). Le `fscanf` réussira dans les deux cas, mais dans le premier cas, la chaîne affectée au $param_i$ sera `"jean dupond"` et dans le second cas, ce sera `"    jean dupond"`.

5.5.6 Entrées formattées : `scanf`

Nous avons déjà vu `scanf`, nous allons la définir ici formellement.

Utilisation

La fonction `scanf` admet un nombre variable de paramètres. Son utilisation est la suivante :

`scanf (` *format* , $param_1$, $param_2$, ... , $param_n$ `)`

Description

Un appel `scanf(fmt,...)` est rigoureusement identique à `fscanf(stdin,fmt,...)`.

5.5.7 Entrées formattées depuis une chaîne : `sscanf`

Utilisation

La fonction `sscanf` admet un nombre variable de paramètres. Son utilisation est la suivante :

`sscanf (` *chaîne* , *format* , $param_1$, $param_2$, ... , $param_n$ `)`

Description

La fonction `sscanf` réalise le même traitement que la fonction `fscanf`, avec la différence que les caractères lus par `sscanf` ne sont pas lus depuis un fichier, mais du tableau de caractères *chaîne*. La rencontre du *null* terminal de *chaîne* pour `sscanf` est équivalent à la rencontre de fin de fichier pour `fscanf`.

5.6 Récréation

En illustration du `printf` et en guise de récréation, je propose un programme dont l'exécution imprime le source du programme. Ce n'est pas facile du tout de créer un tel programme si on exclut la version triviale consistant à faire un `open` sur le source, le lire et l'imprimer. Voici une solution possible que l'on trouve dans le source du compilateur GNU C :

```
main(){char*p="main(){char*p=%c%s%c;printf(p,34,p,34,10);}%c";printf(p,34,p,34,10);}
```

Une indication pour le comprendre : 34 est le code ASCII de " et 10 est le code ASCII de *newline*.

5.7 Exercice 1

Soit un fichier de données structuré en une suite de lignes contenant chacune un nom de personne, un nom de pièce, un nombre et un prix. Exemple :
dupond vilebrequin 10 1000
écrire une procédure `main` dans laquelle on déclarera les variables suivantes :

- nom et article : tableaux de 80 caractères
- nombre et prix : entiers

le corps de la procédure consistera en une boucle dont chaque itération lira une ligne et l'imprimera.

- la lecture d'une ligne se fera par un appel à scanf affectant les 4 champs de la ligne aux 4 variables nom, article, nombre et prix.
- l'écriture consistera à imprimer nom, article et le produit nombre×prix.

5.8 Exercice 2

Reprendre la calculette réalisée en fin de chapitre sur les pointeurs et y rajouter une gestion correcte des erreurs. Si l'utilisateur tape une ligne incorrecte, on désire l'émission d'un message d'erreur, et une continuation de la boucle de calcul.

```c
#include "stdio.h"

/*****************************************************************************/
/*                                                                           */
/*                              main                                         */
/*                                                                           */
/*****************************************************************************/
int main()
{
FILE * fi;
char nom[80];
char article[80];
int nombre,prix;

if ((fi = fopen("exer6.data","r")) == NULL)
   printf("Impossible d'ouvrir le fichier exer6.data\n");
else
   {
   while(fscanf(fi,"%s %s %d %d",nom,article,&nombre,&prix) != EOF)
      printf("%s %s %d\n",nom,article,nombre * prix);
   fclose(fi);
   }
}
```

```c
#include <stdio.h>

enum {FAUX, VRAI};

/*****************************************************************************/
/*                                                                           */
/*                                main                                       */
/*                                                                           */
/*****************************************************************************/
int main()
{
int i,j,r;  /*   les opérandes  */
char c;     /*   l'opérateur   */
char imp;   /*   booléen de demande d'impression du résultat  */
int ret;    /*   code de retour de scanf   */
char buf_err[80];

while (1)
   {
   if ((ret = scanf("%d %c %d",&i,&c,&j)) != 3)
      {
      if (ret == EOF) exit(0);
      scanf("%[^\n]",buf_err);   /*   on mange la partie erronée   */
      printf("Erreur de syntaxe : %s\n",buf_err);
      continue;
      }
   imp = VRAI;
   switch (c)
      {
      case '+' : r = i + j; break;
      case '-' : r = i - j; break;
      case '*' : r = i * j; break;
      case '/' :
         if ( j == 0)
            {
            printf("Division par zéro\n");
            imp = FAUX;
            }
         else r = i / j;
         break;
      case '%' : r = i % j; break;
      default  : printf("l'opérateur %c est incorrect\n",c); imp = FAUX;
      }  /*   fin du switch   */

   if (imp) printf("%d\n",r);
   }
}
```

Chapitre 6

Structures, unions et énumérations

6.1 Notion de structure

Il est habituel en programmation d'avoir besoin d'un mécanisme permettant de grouper un certain nombre de variables de types différents au sein d'une même entité. On travaille par exemple sur un fichier de personnes et on voudrait regrouper une variable de type chaîne de caractères pour le nom, une variable de type entier pour le numéro d'employé, etc. La réponse à ce besoin est le concept d'*enregistrement* : un enregistrement est un ensemble d'éléments de types différents repérés par un nom. Les éléments d'un enregistrement sont appelés des *champs*. Le langage C possède le concept d'enregistrement avec cependant un problème de vocabulaire :

ce que tout le monde appelle	le langage C l'appelle
enregistrement	structure
champ d'un enregistrement	membre d'une structure

6.2 Déclaration de structure

Il y a plusieurs méthodes possibles pour déclarer des structures.

Première méthode

La déclaration :

```
struct personne
   {
   char nom[20];
   char prenom[20];
   int no_employe;
   };
```

déclare l'identificateur `personne` comme étant le nom d'un type de structure composée de trois membres, dont le premier est un tableau de 20 caractères nommé `nom`, le second un tableau de 20 caractères nommé `prenom`, et le dernier un entier nommé `no_employe`. Dans le jargon du langage C, l'identificateur `personne` est une *étiquette de structure*. On peut ensuite utiliser ce type structure pour déclarer des variables, de la manière suivante :

```
struct personne p1,p2;
```

qui déclare deux variables de type `struct personne` de noms `p1` et `p2`;

Deuxième méthode

On peut déclarer des variables de type structure sans utiliser d'étiquette de structure, par exemple :

```
struct
   {
   char nom[20];
   char prenom[20];
   int no_employe;
   } p1,p2;
```

déclare deux variables de noms `p1` et `p2` comme étant deux structures de trois membres, mais elle ne donne pas de nom au type de la structure. L'inconvénient de cette méthode est qu'il sera par la suite impossible de déclarer une autre variable du même type. En effet, si plus loin on écrit :

```
struct
   {
   char nom[20];
   char prenom[20];
   int no_employe;
   } p3;
```

les deux structures ont beau avoir le même nombre de champs, avec les mêmes noms et les mêmes types, elles seront considérées de types différents. Il sera impossible en particulier d'écrire `p3 = p1;`.

Troisième méthode

On peut combiner déclaration d'étiquette de structure et déclaration de variables, comme ceci :

```
struct personne
   {
   char nom[20];
   char prenom[20];
   int no_employe;
   } p1,p2;
```

déclare les deux variables `p1` et `p2` et donne le nom `personne` à la structure. Là aussi, on pourra utiliser ultérieurement le nom `struct personne` pour déclarer d'autres variables :

```
struct personne pers1, pers2, pers3;
```

qui seront du même type que `p1` et `p2`.

De ces trois méthodes c'est la première qui est recommandée, car elle permet de bien séparer la définition du type structure de ses utilisations.

Initialisation d'une structure

Une structure peut être initialisée par une liste d'expressions constantes à la manière des initialisations de tableau. Exemple :

```
struct personne p = {"Jean", "Dupond", 7845};
```

6.3 Opérateurs sur les structures

6.3.1 Accès aux membres des structures

Pour désigner un membre d'une structure, il faut utiliser l'opérateur de sélection de membre qui se note `.` (point). Par exemple, si `p1` et `p2` sont deux variables de type `struct personne`, on désignera le membre `nom` de `p1` par `p1.nom` et on désignera le membre `no_employe` de `p2` par `p2.no_employe`. Les membres ainsi désignés se comportent comme n'importe quelle variable et par exemple, pour accéder au premier caractère du nom de `p2`, on écrira : `p2.nom[0]`.

6.3.2 Affectation de structures

On peut affecter une structure à une variable structure de même type, grâce à l'opérateur d'affectation :

```
struct personne p1,p2
...
p1 = p2;
```

6.3.3 Comparaison de structures

Aucune comparaison n'est possible sur les structures, même pas les opérateurs `==` et `!=`.

6.4 Tableaux de structures

Une déclaration de tableau de structures se fait selon le même modèle que la déclaration d'un tableau dont les éléments sont de type simple. Supposons que l'on ait déjà déclaré la `struct personne`, si on veut déclarer un tableau de 100 structures de ce type, on écrira :

```
struct personne t[100];
```

Pour référencer le nom de la personne qui a l'index `i` dans `t` on écrira : `t[i].nom`.

6.5 Exercice

Soit un fichier de données identiques à celui de l'exercice précédent.
Écrire une procédure `main` qui :

1. lise le fichier en mémorisant son contenu dans un tableau de structures, chaque structure permettant de mémoriser le contenu d'une ligne (nom, article, nombre et prix).

2. parcoure ensuite ce tableau en imprimant le contenu de chaque structure.

```
#include <stdio.h>

/*****************************************************************************/
/*                              main                                         */
/*****************************************************************************/
int main()
{
FILE * fi;
struct commande
   {
   char nom[80];
   char article[80];
   int nombre,prix;
   };

#define nb_com 100
struct commande tab_com[nb_com];   /*   tableau des commandes   */

int i;   /*   index dans tab_com   */
int ilast;   /*   dernier index valide dans tab_com après remplissage   */

if ((fi = fopen("exer7.data","r")) == NULL)
   printf("Impossible d'ouvrir le fichier exer7.data\n");
else
   {
   /*   boucle de lecture des commandes   */
   /*   -------------------------------   */
   i = 0;

   while(i < nb_com && fscanf(fi,"%s %s %d %d",
                          tab_com[i].nom,
                          tab_com[i].article,
                          &tab_com[i].nombre,
                          &tab_com[i].prix) != EOF)
      i++;   /*   corps du while */

   if (i >= nb_com)
      printf("le tableau tab_com est sous-dimentionné\n");
   else
      {
      /*   impression des commandes mémorisées   */
      /*   -----------------------------------   */
      ilast = i - 1;

      for (i = 0; i <= ilast; i++)
         printf("%s %s %d %d\n", tab_com[i].nom, tab_com[i].article,
                                tab_com[i].nombre, tab_com[i].prix);

      fclose(fi);
      }
   }
}
```

6.6 Pointeurs vers une structure

Supposons que l'on ait défini la `struct personne` à l'aide de la déclaration :

```
struct personne
   {
   ...
   };
```

on déclarera une variable de type pointeur vers une telle structure de la manière suivante :

```
struct personne *p;
```

on pourra alors affecter à `p` des adresses de `struct personne`. Exemple :

```
struct personne
   {
   ...
   };

int main()
{
struct personne pers;   /*   pers est une variable de type struct personne   */
struct personne *p;     /*   p est un pointeur vers une struct personne      */

p =  &pers;
}
```

6.7 Structures dont un des membres pointe vers une structure du même type

Une des utilisations fréquentes des structures, est de créer des listes de structures chaînées. Pour cela, il faut que chaque structure contienne un membre qui soit de type pointeur vers une structure du même type. Cela se fait de la façon suivante :

```
struct personne
   {
   ...  /*   les différents membres   */
   struct personne *suivant;
   };
```

le membre de nom `suivant` est déclaré comme étant du type pointeur vers une `struct personne`. La dernière structure de la liste devra avoir un membre `suivant` dont la valeur sera le pointeur `NULL` que nous avons vu en 5.1

6.8 Accès aux éléments d'une structure pointée

Supposons que nous ayons déclaré `p` comme étant de type pointeur vers une `struct personne`, comment écrire une référence à un membre de la structure pointée par `p` ?

Étant donné que *p désigne la structure, on serait tenté d'écrire *p.nom pour référencer le membre nom. Mais il faut savoir que les opérateurs d'indirection (*) et de sélection (.), tout comme les opérateurs arithmétiques, ont une priorité. Et il se trouve que l'indirection a une priorité inférieure à celle de la sélection. Ce qui fait que *p.nom sera interprété comme signifiant *(p.nom). (Cela aurait un sens si p était une structure dont un des membres s'appelait nom et était un pointeur). Dans notre cas, il faut écrire (*p).nom pour forcer l'indirection à se faire avant la sélection.

Cette écriture étant assez lourde, le langage C a prévu un nouvel opérateur noté -> qui réalise à la fois l'indirection et la sélection : p -> nom est identique à (*p).nom. Exemple : si p est de type pointeur vers la struct personne définie précédemment, pour affecter une valeur au membre no_employe de la structure pointée par p, on peut écrire :

```
p -> no_employe = 13456;
```

6.9 Passage de structures en paramètre

Supposons que l'on ait fait la déclaration suivante :

```
struct date
   {
   int jour,mois,annee;
   };
```

une fonction de comparaison de deux dates pourra s'écrire :

```
enum {AVANT, EGAL, APRES};

int cmp_date( struct date d1, struct date d2)
{
if (d1.annee > d2.annee)
   return(APRES);
if (d1.annee < d2.annee)
   return(AVANT);
...   /*   comparaison portant sur mois et jour   */
}
```

et une utilisation de cette fonction pourra être :

```
struct date d1,d2;

if (cmp_date(d1,d2) == AVANT)
   ...
```

Attention

En langage C K&R, il n'est pas possible de passer en paramètre une structure, mais on peut passer un pointeur vers une structure.

6.10 Détermination de la taille allouée à un type

Pour connaître la taille en octets de l'espace mémoire nécessaire pour une variable, on dispose de l'opérateur `sizeof`. Cet opérateur est un opérateur unaire préfixé que l'on peut employer de deux manières différentes : soit `sizeof` *expression* soit `sizeof (` *nom-de-type* `)`. Exemple :

```
int i,taille;

taille = sizeof i;
taille = sizeof (short int);
taille = sizeof (struct personne);
```

6.10.1 Retour sur la conversion des tableaux

L'opérande de l'opérateur `sizeof` est la seule exception à la conversion d'un identificateur de type tableau de `x` en pointeur vers `x`. Ne pas réaliser cette conversion est en effet nécessaire pour que l'opérateur `sizeof` ait l'effet attendu par le programmeur lorsqu'il l'applique à un tableau. Exemple :

```
int t[10];

if (sizeof(t) / sizeof(int) != 10)
   printf("sizeof mal implémenté\n");
else printf("sizeof ok\n");
```

6.11 Allocation et libération d'espace pour les structures

Nous allons voir dans ce paragraphe trois fonctions de la bibliothèque standard permettant d'allouer et de libérer de l'espace.

6.11.1 Allocation d'espace : fonctions `malloc` et `calloc`

Quand on crée une liste chaînée, c'est parce qu'on ne sait pas à la compilation combien elle comportera d'éléments à l'exécution (sinon on utiliserait un tableau). Pour pouvoir créer des listes, il est donc nécessaire de pouvoir allouer de l'espace dynamiquement. On dispose pour cela de deux fonctions `malloc` et `calloc`.

Allocation d'un élément : fonction `malloc`

La fonction `malloc` admet un paramètre qui est la taille en octets de l'élément désiré et elle rend un pointeur vers l'espace alloué. Utilisation typique :

```
#include <stdlib.h>
struct personne *p;

p = malloc(sizeof(struct personne));
```

Allocation d'un tableau d'éléments : fonction `calloc`

Elle admet deux paramètres :

- le premier est le nombre d'éléments désirés ;
- le second est la taille en octets d'un élément.

son but est d'allouer un espace suffisant pour contenir les éléments demandés et de rendre un pointeur vers cet espace. Utilisation typique :

```
#include <stdlib.h>
struct personne *p;
int nb_elem;

...     /*   init  de nb_elem   */
p = calloc(nb_elem,sizeof(struct personne));
```

On peut alors utiliser les éléments `p[0]`, `p[1]`, ... `p[nb_elem-1]`.

6.11.2 Libération d'espace : procédure free

On libère l'espace alloué par `malloc` ou `calloc` au moyen de la procédure `free` qui admet un seul paramètre : un pointeur précédemment rendu par un appel à `malloc` ou `calloc`. Utilisation typique :

```
#include <stdlib.h>
struct personne *p;

p = malloc(sizeof(struct personne));
...   /*   utilisation de la structure allouée   */
free(p);
```

6.12 Exercice

Modifier le programme précédent :

1. en écrivant une procédure d'impression d'une `struct commande` passée en paramètre.
2. en écrivant une fonction de recherche de commande maximum (celle pour laquelle le produit nombre × prix est maximum). Cette fonction admettra en paramètre un pointeur vers la `struct commande` qui est tête de la liste complète, et rendra un pointeur vers la structure recherchée.
3. le `main` sera modifié de manière à faire appel à la fonction de recherche de la commande maximum et à imprimer cette commande.

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

/*   les types communs a toutes les procedures   */
/*   ---------------------------------------     */
struct commande
   {
   char nom[80];
   char article[80];
   int nombre,prix;
   struct commande *suiv;
   };

/******************************************************************************/
/*                                                                            */
/*                             print_com                                      */
/*                                                                            */
/*   But:                                                                     */
/*      Imprime une structure commande                                        */
/*                                                                            */
/******************************************************************************/
void print_com(struct commande com)
{
printf("%s %s %d %d\n",
       com.nom, com.article, com.nombre, com.prix);
}
```

```
/*****************************************************************************/
/*                                                                           */
/*                              max_com                                      */
/*                                                                           */
/*   But:                                                                    */
/*      Recherche la commande pour laquelle le produit nombre * prix est     */
/*      le maximun                                                           */
/*                                                                           */
/*   Interface:                                                              */
/*      l_com : la liste dans laquelle doit se faire la recherche            */
/*      valeur rendue : pointeur vers la structure commande recherchée       */
/*                      ou NULL si l_com est vide                            */
/*                                                                           */
/*****************************************************************************/
struct commande *max_com(struct commande * l_com)
{
struct commande *pmax;   /*   pointeur vers le max courant   */
struct commande *pcour;  /*   pointeur vers l'element courant   */
int vmax,vcour;

if (l_com == NULL)
   return(NULL);
else
   {
   pmax = l_com; vmax = (pmax -> nombre) * (pmax -> prix);

   for (pcour = l_com -> suiv; pcour != NULL; pcour = pcour ->suiv)
      {
      vcour = (pcour -> nombre * pcour -> prix);
      if (vcour > vmax)
         {
         vmax = vcour;
         pmax = pcour;
         }
      }

   return(pmax);
   }
}
```

```
/*****************************************************************************/
/*                               main                                        */
/*****************************************************************************/
int main()
{
FILE * fi;
struct commande *l_com = NULL; /*   liste des commandes                          */
struct commande *prec,*cour;   /*   pour la commande précédente et courante      */
int val_ret;                   /*   valeur de retour de fscanf                   */

if ((fi = fopen("exer7.data","r")) == NULL)
   printf("Impossible d'ouvrir le fichier exer7.data\n");
else
   {
   /*   lecture du fichier avec création de la liste de commandes   */
   /*   --------------------------------------------------------   */
   do
      {
      cour = malloc(sizeof(struct commande));
      val_ret = fscanf(fi,"%s %s %d %d", cour -> nom, cour -> article,
                       &(cour -> nombre), &(cour -> prix));
      if (val_ret == EOF)
         {
         free(cour);
         if(l_com != NULL) prec -> suiv = NULL;
         }
      else
         {
         if (l_com == NULL) l_com = cour; else prec -> suiv = cour;
         prec = cour;
         }
      }
   while (val_ret != EOF);

   /*   parcours de la liste avec impression   */
   /*   ------------------------------------   */
   if (l_com == NULL)
      printf("La liste de commandes est vide\n");
   else
      {
      for (cour = l_com; cour != NULL; cour = cour -> suiv)
         print_com(*cour);

      /*   recherche et impression de la commande maximum   */
      /*   ----------------------------------------------   */
      printf("La commande maximum est :\n");
      print_com(*max_com(l_com));
      }

   fclose(fi); /*   fermeture du fichier   */
   }
}
```

6.13 Les champs de bits

6.13.1 Généralités

Il est parfois nécessaire pour un programmeur de décrire en termes de bits la structure d'une ressource matérielle de la machine. Un exemple typique est la programmation système qui nécessite de manipuler des registres particuliers de la machine. Par exemple, dans le manuel du MC 68030 de Motorola, le registre d'état est ainsi décrit :

- bit 0 : carry ;
- bit 1 : overflow ;
- bit 2 : zéro ;
- bit 3 : négatif ;
- bit 4 : extension ;
- bits 5-7 : inutilisés ;
- bits 8-10 : masque des interruptions ;
- bit 11 : inutilisé ;
- bits 12-13 : niveau de privilège ;
- bits 14-15 : état des traces.

Il existe dans le langage C un moyen de réaliser de telles descriptions, à l'aide du concept de structure. En effet, dans une déclaration de structure, il est possible de faire suivre la définition d'un membre par une indication du nombre de bits que doit avoir ce membre. Dans ce cas, le langage C appelle ça un *champ de bits*.

Le registre d'état du MC 68030 peut se décrire ainsi :

```
struct sr
   {
   unsigned int trace : 2;
   unsigned int priv : 2;
   unsigned int : 1;            /*   inutilisé   */
   unsigned int masque : 3;
   unsigned int : 3;            /*   inutilisé   */
   unsigned int extend : 1;
   unsigned int negative : 1;
   unsigned int zero : 1;
   unsigned int overflow : 1;
   unsigned int carry : 1;
   };
```

On voit que le langage C accepte que l'on ne donne pas de nom aux champs de bits qui ne sont pas utilisés.

6.13.2 Contraintes

1. Les seuls types acceptés pour les champs de bits sont `int`, `unsigned int` et `signed int`.
2. L'ordre dans lequel sont mis les champs de bits à l'intérieur d'un mot dépend de l'implémentation, mais généralement, dans une machine *little endian* les premiers champs décrivent les bits de poids faibles et les derniers champs les bits de poids forts, alors que c'est généralement l'inverse dans une machine *big endian*.
3. Un champ de bit déclaré comme étant de type `int`, peut en fait se comporter comme un `signed int` ou comme un `unsigned int` (cela dépend de l'implémentation). Il est donc recommandé d'une manière générale de déclarer les champs de bits comme étant de type `unsigned int`.
4. Un champ de bits n'a pas d'adresse, on ne peut donc pas lui appliquer l'opérateur adresse de (`&`).

6.14 Les énumérations

Nous avons vu au paragraphe 1.10.2 que l'on pouvait déclarer des constantes nommées de la manière suivante :

```
enum {LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE};
```

qui déclare les identificateurs `LUNDI`, `MARDI`, etc. comme étant des constantes entières de valeur 0, 1, etc. Ce qui n'avait pas été dit à ce moment là, c'est que les énumérations fonctionnent syntaxiquement comme les structures : après le mot-clé `enum` il peut y avoir un identificateur appelé *étiquette d'énumération* qui permettra plus loin dans le programme de déclarer des variables de type énumération. Exemple :

```
enum jour {LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE};

enum jour j1, j2;
j1 = LUNDI;
j2 = MARDI;
```

L'exemple ci-dessus est conforme à ce qui nous semble être de bonnes règles de programmation : déclarer d'abord le type énumération en lui donnant un nom grâce à une étiquette d'énumération et ensuite utiliser ce nom pour déclarer des variables. Cependant, comme pour les structures, le langage C permet de déclarer des variables dans la déclaration du type énumération, éventuellement en omettant l'étiquette d'énumération. Exemples :

```
enum jour {LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE} d1, d2;
```

déclare `d1` et `d2` comme étant des variable de type `enum jour`,

```
enum {FAUX, VRAI} b1,b2;
```

déclare `b1` et `b2` comme étant des variables de type énumération sans nom, dont les valeurs peuvent être `FAUX` ou `VRAI`. Nous avons expliqué plus haut pourquoi un tel style de programmation nous semblait mauvais.

6.15 Les unions

Il est parfois nécessaire de manipuler des variables auxquelles on désire affecter des valeurs de type différents. Supposons que l'on désire écrire un package mathématique qui manipulera des nombres qui seront implémentés par des `int`, tant que la précision des entiers de la machine sera suffisante et qui passera automatiquement à une représentation sous forme de flottants dès que ce ne sera plus le cas. Il sera nécessaire de disposer de variables pouvant prendre soit des valeurs entières, soit des valeurs flottantes.

Ceci peut se réaliser en C, grâce au mécanisme des unions. Une définition d'union a la même syntaxe qu'une définition de structure, le mot clé `struct` étant simplement remplacé par le mot clé `union`. Exemple :

```
union nombre
   {
   int i;
   float f;
   }
```

L'identificateur `nombre` est appelé *étiquette d'union*. La différence sémantique entre les `struct` et les `unions` est la suivante : alors que pour une variable de type structure tous les membres peuvent avoir en même temps une valeur, une variable de type union ne peut avoir à un instant donné qu'un seul membre ayant une valeur. En reprenant l'exemple précédent, on déclarera une variable `n` de type `union nombre` par :

```
union nombre n;
```

cette variable pourra posséder soit une valeur entière, soit une valeur flottante, mais pas les deux à la fois.

6.16 Accès aux membres de l'union

Cet accès se fait avec le même opérateur sélection (noté `.`) que celui qui sert à accéder aux membres des structures. Dans l'exemple précédent, si on désire faire posséder à la variable `n` une valeur entière, on écrira :

```
n.i = 10;
```

si on désire lui faire posséder une valeur flottante, on écrira :

```
n.f = 3.14159;
```

6.17 Utilisation pratique des unions

Lorsqu'il manipule des variables de type union, le programmeur n'a malheureusement aucun moyen de savoir à un instant donné, quel est le membre de l'union qui possède une valeur. Pour être utilisable, une union doit donc toujours être associée à une variable dont le but sera d'indiquer le membre de l'union qui est valide. En pratique, une union et son indicateur sont généralement englobés à l'intérieur d'une structure. Dans l'exemple précédent, on procédera de la manière suivante :

```
enum type {ENTIER, FLOTTANT};

struct arith
   {
   enum type typ_val;   /*   indique ce qui est dans u   */
   union
      {
      int i;
      float f;
      } u;
   };
```

la `struct arith` a deux membres `typ_val` de type `int`, et `u` de type union d'`int` et de `float`. On déclarera des variables par :

```
struct arith a1,a2;
```

puis on pourra les utiliser de la manière suivante :

```
a1.typ_val = ENTIER;
a1.u.i = 10;

a2.typ_val = FLOTTANT;
a2.u.f = 3.14159;
```

Si on passe en paramètre à une procédure un pointeur vers une `struct arith`, la procédure testera la valeur du membre `typ_val` pour savoir si l'union reçue possède un entier ou un flottant.

6.18 Une méthode pour alléger l'accès aux membres

Quand une union est dans une structure, il faut donner un nom au membre de la structure qui est de type union, ce qui à pour conséquence de rendre assez lourd l'accès aux membres de l'union. Dans l'exemple précédent, il faut écrire `a1.u.f` pour accéder au membre `f`. On peut alléger l'écriture en utilisant les facilités du préprocesseur. On peut écrire par exemple :

```
#define I u.i
#define F u.f
```

Pour initialiser `a1` avec l'entier 10, on écrira alors :

```
a1.typ_val = ENTIER;
a1.I = 10;
```

Chapitre 7

Les expressions

Ce chapitre débute par l'étude des conversions, problème qui avait été à peine effleuré quand nous avions parlé des opérateurs. Il se poursuit par la présentation des opérateurs non encore vus et se termine par l'étude de la sémantique des expressions.

7.1 Les conversions de types

7.1.1 Utilité des conversions

Dans un programme, dans un contexte où l'on attend une valeur d'un certain type, il faut normalement fournir une valeur de ce type. Par exemple, si la partie gauche d'une affectation est de type flottant, la valeur fournie en partie droite doit également être de type flottant. Il est cependant agréable de ne pas être trop strict sur cette règle. Si le type attendu et le type de la valeur fournie sont trop différents, (on attend un flottant et on fournit une structure), il est normal que le compilateur considère qu'il s'agit d'une erreur du programmeur. Si par contre, le type attendu et le type de la valeur fournie sont assez « proches », c'est une facilité agréable que le compilateur fasse lui-même la conversion. On peut admettre par exemple, que dans un contexte où on attend un nombre flottant, on puisse fournir un nombre entier.

Autre situation où les conversions sont utiles : les expressions. Les machines physiques sur lesquelles s'exécutent les programmes comportent des instructions différentes pour réaliser de l'arithmétique sur les entiers et sur les flottants. Cette situation se retrouve dans les langages de programmation de bas niveau (les assembleurs) où le programmeur doit utiliser des opérateurs différents pour réaliser la même opération (au sens mathématique du terme) selon qu'elle porte sur des entiers ou des flottants. Les langages de programmation de haut niveau par contre, surchargent les symboles des opérateurs arithmétiques de manière à ce que le même symbole puisse réaliser une opération indifféremment entre entiers ou entre flottants : le symbole `+` permet de réaliser l'addition de deux entiers ou deux flottants. Ceci est déjà une facilité agréable, mais il est possible d'aller plus loin. Le langage peut autoriser le programmeur à donner aux opérateurs des opérandes de types différents, charge au compilateur de faire une conversion de type sur l'un ou l'autre des opérandes pour les amener à un type commun.

Enfin, il se peut que le langage offre au programmeur la possibilité de demander explicitement une conversion de type : si le langage PASCAL n'offre pas une telle possibilité, le langage C par contre dispose d'un opérateur de conversion de type.

7.1.2 Ce qu'il y a dans une conversion

Pour comprendre ce qui se passe dans une conversion il faut bien distinguer type, valeur et représentation. La représentation d'une valeur est la chaîne de bits qui compose cette valeur dans la mémoire de la machine. La représentation des entiers est une suite de bits en notation binaire simple pour les positifs, généralement en complément à 2 pour les négatifs. La représentation des flottants est plus compliquée, c'est généralement un triplet de chaînes de bits : (signe, mantisse, exposant).

Une conversion a pour but de changer le type d'une valeur, sans changer cette valeur si c'est possible ; elle pourra éventuellement s'accompagner d'un changement de représentation.

Exemple de conversion avec changement de représentation : la conversion d'entier vers flottant ou vice versa. Exemple de conversion sans changement de représentation : la conversion d'entier non signé vers entier signé ou vice versa, sur une machine où les entiers signés sont représentés en complément à 2.

7.1.3 L'ensemble des conversions possibles

Conversions vers un type entier

- **depuis un type entier** La règle est de préserver, si c'est possible, la valeur mathématique de l'objet. Si ce n'est pas possible :

  - si le type destination est un type signé, on considère qu'il y a dépassement de capacité et la valeur du résultat n'est pas définie.
  - si le type destination est un type non signé, la valeur du résultat doit être égale (modulo n) à la valeur originale, où n est le nombre de bits utilisés pour représenter les valeur du type destination.

  Dans ce qui suit, on se place précisément dans la cas où la machine représente les nombres signés en complément à 2 (c'est le cas de pratiquement toutes les machines). Une conversion d'un entier signé vers un entier non signé, ou vice versa, se fait sans changement de représentation. Une conversion d'un entier vers un entier plus court se fait par troncature des bits les plus significatifs. Une conversion d'un entier vers un entier plus long se fait par extension du bit de signe si le type originel est signé, par extension de zéros si le type originel est non signé.

- **depuis un type flottant** La règle est de préserver, si c'est possible, la valeur mathématique de l'objet, sachant qu'il peut y avoir une erreur d'arrondi.

- **depuis un pointeur** Un pointeur peut être converti en un type entier. Pour cela il est considéré comme un type entier non signé de la même taille que les pointeurs. Il est ensuite converti dans le type destination selon les règles de conversions d'entiers vers entiers.

Conversions vers un type flottant

Seuls les types entiers et flottants peuvent être convertis en un type flottant. Là aussi, la règle est de préserver la valeur si possible, sinon c'est un cas d'overflow ou d'underflow.

Conversion vers un type pointeur

Les différentes possibilités sont les suivantes :

- Un type pointeur vers $T1$ peut être converti en un type pointeur vers $T2$ quels que soient $T1$ et $T2$.
- La valeur entière 0 peut être convertie en un type pointeur vers T quel que soit T, et c'est la valeur dite de *pointeur invalide*.
- Une valeur entière non nulle peut être convertie en un type pointeur vers T quel que soit T, mais cela est explicitement non portable.

Nous avons vu précédemment au paragraphe 4.1 :

- Toute expression de type tableau de `x` est convertie en type pointeur vers `x`.

Il y a une règle similaire concernant les fonctions :

- Toute expression de type fonction retournant `x` est convertie en type pointeur vers fonction retournant `x`.

Conversion vers le type void

N'importe quelle valeur peut être convertie vers le type void. Cela n'a de sens que si la valeur résultat n'est pas utilisée.

7.1.4 Les situations de conversions

Dans le langage C, les situations où se produisent les conversions sont les suivantes :

1. une valeur d'un certain type est utilisée dans un contexte qui en demande un autre.
   - passage de paramètre : le paramètre effectif n'a pas le type du paramètre formel ;
   - affectation : la valeur à affecter n'a pas le même type que la variable ;
   - valeur rendue par une fonction : l'opérande de *return* n'a pas le type indiqué dans la déclaration de la fonction.
2. opérateur de conversion : le programmeur demande explicitement une conversion.
3. un opérateur a des opérandes de types différents.

Dans les cas 1 et 2, type de départ et type d'arrivée de la conversion sont donnés. Dans le cas 3, par contre, c'est le compilateur qui choisit le type d'arrivée de la conversion. Il le fait selon des règles soigneusement définies. Il y en a deux dans le langage C qui portent les noms de « promotion des entiers » et « conversions arithmétiques habituelles ».

7.1.5 La promotion des entiers

Ce que l'on appelle dans le langage C *promotion des entiers* est une règle de conversion des opérandes dans les expressions. La promotion des entiers a pour but d'amener les « petits entiers » à la taille des `int`.

Domaine d'application

La promotion des entiers est appliquée à l'opérande des opérateurs unaires `+`, `-` et `~`, ainsi qu'aux deux opérandes des opérateurs de décalage `>>` et `<<`. La promotion des entiers est également utilisée dans la définition des *conversions arithmétiques habituelles*.

La règle

Une valeur de type `char`, un `short int` ou un champ de bits, ou d'une version signée ou non signée des précédents, peut être utilisée dans un contexte où `un int` ou un `unsigned int` est demandé. Cette valeur est convertie en `un int` ou un `unsigned int` d'une manière (hélas) dépendante de l'implémentation :

- si un `int` peut représenter toutes les valeurs du type de départ, la valeur est convertie en `int` ;
- sinon, elle est convertie en `unsigned int`.

7.1.6 Les conversions arithmétiques habituelles

Domaine d'application

Les conversions arithmétiques habituelles sont réalisés sur les opérandes de tous les opérateurs arithmétiques binaires sauf les opérateurs de décalage `>>` et `<<` ainsi que sur les second et troisième opérandes de l'opérateur `?:`.

La règle

1. Si un opérande est de type `long double`, l'autre opérande est converti en `long double`.
2. Sinon si un opérande est de type `double`, l'autre opérande est converti en `double`.
3. Sinon si un opérande est de type `float`, l'autre opérande est converti en `float`.
4. Sinon la promotion des entiers est réalisée sur les deux opérandes. Ensuite :
   a. Si un opérande est de type `unsigned long int`, l'autre opérande est converti en `unsigned long int`.
   b. Sinon, si un opérande est de type `long int` et l'autre de type `unsigned int`, alors :
      - si un `long int` peut représenter toutes les valeurs d'un `unsigned int`, l'opérande de type `unsigned int` est converti en `long int`.
      - sinon, les deux opérandes sont convertis en `unsigned long int`.
   c. Sinon, si un opérande est de type `long int`, l'autre opérande est converti en `long int`.
   d. Sinon, si un opérande est de type `unsigned int`, l'autre opérande est converti en `unsigned int`.
   e. Sinon, les deux opérandes sont de même type, et il n'y a pas de conversion à réaliser.

Discussion

Les points 1, 2, 3 sont faciles à comprendre : si les deux opérandes sont flottants, celui de moindre précision est converti dans le type de l'autre. Si un seul des opérandes est de type flottant, l'autre est converti dans ce type.

On aborde le point 4 si les deux opérandes sont des variétés d'entiers courts, normaux ou longs, signés ou non signés. On applique alors la promotion des entiers, de manière à se débarrasser des entiers courts. À la suite de cela, il n'y plus comme types possibles que `int`, `unsigned int`, `long int` et `unsigned long int`.

Si l'on excepte les cas où les deux types sont identiques, le reste des règles peut se résumer dans le tableau suivant :

opérande	opérande	résultat
`unsigned long int`	*quelconque*	`unsigned long int`
`long int`	`unsigned int`	`long int` `unsigned long int`
`long int`	`int`	`long int`
`unsigned int`	`int`	`unsigned int`

7.1.7 Les surprises des conversions

D'une manière générale, les conversions sont un mécanisme qui fonctionne à la satisfaction du programmeur. Il y a cependant une situation où cela peut donner des résultats surprenants : quand on réalise une comparaison entre entiers signés et entiers non signés. Par exemple, le programme suivant :

```
int main()
{
unsigned int i = 0;

if (i < -1 )
   printf("Bizarre, bizarre ...\n");
   else printf ("Tout semble normal\n");
}
```

imprimera le message `Bizarre, bizarre ...`, pouvant laisser croire que pour le langage C, 0 est inférieur à -1.

L'explication est la suivante : l'opérateur $<$ a un opérande de type `unsigned int` (la variable `i`), et un autre opérande de type `int` (la constante -1). D'après le tableau des conversions donné ci-dessus, on voit que dans un tel cas, les opérandes sont convertis en `unsigned int`. Le compilateur génère donc une comparaison non signée entre 0 et 4294967295 (puisque -1 = 0xffffffff = 4294967295), d'où le résultat.

Pour que tout rentre dans l'ordre, il suffit d'utiliser l'opérateur de conversion pour prévenir le compilateur de ce qu'on veut faire :

```
int main()
{
unsigned int i = 0;
```

```
if ((int) i < -1 )     /*   comparaison entre deux int    */
   printf("Bizarre, bizarre ...\n");
   else printf ("Tout semble normal\n");
}
```

Là où tout se complique c'est qu'on peut utiliser des entiers non signés sans le savoir ! Considérons le programme suivant :

```
int main()
{
if (sizeof(int) < -1)
   printf("Bizarre, bizarre ...\n");
else printf ("Tout semble normal\n");
}
```

le lecteur a sans doute deviné qu'il va imprimer le message `Bizarre, bizarre ...`, et cependant les entiers n'ont pas une longueur négative ! L'explication est la suivante : l'opérateur `sizeof` rend une valeur dont le type est non signé. Voici ce que dit exactement la norme : « *La valeur du résultat* [de `sizeof`] *dépend de l'implémentation, et son type (un type entier non signé) est* `size_t` qui est définit dans le fichier d'include `stddef.h` ». Dans notre exemple, le compilateur a généré une comparaison non signée entre 4 (`sizeof(int)`) et 4 294 967 295, d'où le résultat.

Recommandations

1. Ne jamais mélanger des entiers signés et non signés dans des comparaisons : utiliser l'opérateur de conversion pour amener l'opérateur de comparaison à avoir des opérandes de même type.

2. Bien noter que l'opérateur `sizeof` rend une valeur de type entier non signé.

7.2 Les opérateurs

Nous avons étudié les opérateurs arithmétiques usuels dans le chapitre sur les bases du langage, les opérateurs incrément et décrément dans le chapitre sur les tableaux, les opérateurs d'adresse dans le chapitre sur les pointeurs et les opérateurs de sélection dans le chapitre sur les structures.

Le langage C comporte quelques autres opérateurs que nous allons étudier ci-après.

7.2.1 Opérateur *non bit à bit*

- Syntaxe :

  expression :
  $\Rightarrow$ `~` *expression*

- Sémantique :

  expression est évaluée et doit délivrer une valeur de type entier, l'opération *non bit à bit* est réalisée sur cette valeur, et le résultat obtenu est la valeur de l'expression `~`.

7.2.2 Opérateur *et bit à bit*

- Syntaxe :

  expression :
  $\Rightarrow$ $expression_1$ `&` $expression_2$

- Sémantique :

  Les deux expressions sont évaluées et doivent délivrer des valeurs de type entier, le *et bit à bit* est réalisé, et la valeur obtenue est la valeur de l'expression `&`.

7.2.3 Opérateur *ou bit à bit*

- Syntaxe :

  expression :
  $\Rightarrow$ $expression_1$ `|` $expression_2$

- Sémantique :

  Les deux expressions sont évaluées et doivent délivrer des valeurs de type entier, le *ou bit à bit* est réalisé, et la valeur obtenue est la valeur de l'expression `|`.

7.2.4 Opérateur *ou exclusif bit à bit*

- Syntaxe :

  expression :
  $\Rightarrow$ $expression_1$ `^` $expression_2$

- Sémantique :

  Les deux expressions sont évaluées et doivent délivrer des valeurs de type entier, le *ou exclusif bit à bit* est réalisé, et la valeur obtenue est la valeur de l'expression `^`.

7.2.5 Opérateur *décalage à gauche*

- Syntaxe :

  expression :
  $\Rightarrow$ $expression_1$ `<<` $expression_2$

- Sémantique :

  Les deux expressions sont évaluées et doivent délivrer des valeurs de type entier, la valeur de $expression_1$ est décalée à gauche de $expression_2$ bits en remplissant les bits libres avec des zéros. Le résultat obtenu est la valeur de l'expression `<<`.

7.2.6 Opérateur *décalage à droite*

- Syntaxe :

  expression :
  $\Rightarrow$ $expression_1$ `>>` $expression_2$

- Sémantique :

  Les deux expressions sont évaluées et doivent délivrer des valeurs de type entier, la valeur de $expression_1$ est décalée à droite de $expression_2$ bits. Si $expression_1$ délivre une valeur `unsigned`, le décalage est un décalage logique : les bits libérés sont remplis avec des zéros. Sinon, le décalage peut être logique ou arithmétique (les bits libérés sont remplis avec le bit de signe), cela dépend de l'implémentation.

7.2.7 Opérateur conditionnel

- Syntaxe :

  expression :
  $\Rightarrow$ $expression_1$? $expression_2$: $expression_3$

- Sémantique :

  $expression_1$ est évaluée et doit délivrer une valeur de type entier. Si cette valeur est :

  - non nulle, $expression_2$ est évaluée et le résultat est la valeur de l'expression conditionnelle.
  - nulle, $expression_3$ est évaluée et le résultat est la valeur de l'expression conditionnelle.

Exemples

Cet opérateur permet de remplacer une instruction `if` :

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

On peut utiliser cet opérateur en cascade, mais la lisibilité en souffre :

```
printf("i est %s",i < 0 ? "negatif\n" : i > 0 ? "positif\n" : "nul\n");
```

7.2.8 Opérateur *virgule*

- Syntaxe :

  expression :
  $\Rightarrow$ $expression_1$, $expression_2$

- Sémantique :

  $expression_1$ est évaluée et sa valeur ignorée. $expression_2$ est évaluée et sa valeur est la valeur de l'expression virgule.

Remarque

Étant donné que la valeur de $expression_1$ est ignorée, pour qu'une telle construction ait un sens, il faut que $expression_1$ fasse un effet de bord. On peut écrire par exemple :

```
i = (j = 2 , 1);
```

ce qui est une manière particulièrement horrible d'écrire :

```
i = 1;
j = 2;
```

Une utilisation agréable par contre de l'opérateur virgule est dans les expressions d'une boucle for. Si on désire écrire une boucle `for` qui utilise deux index, il est utile d'écrire par exemple :

```
for (i = 1, j = 1; i <= LIMITE; i++, j = j + 2)
   {
   ...
   }
```

ceci permet de rendre manifeste que `i = 1` et `j = 1` sont la partie initialisation et `i++` et `j = j + 2` sont la partie itération de la boucle.

7.2.9 Opérateurs d'affectation composée

Chacun des opérateurs `+ - * / % >> << & ^ |` peut s'associer à l'opérateur d'affectation pour former respectivement les opérateurs `+= -= *= /= %= >>= <<= &= ^= |=`.

Nous donnerons la syntaxe et la sémantique de ces opérateurs dans le cas de l'opérateur `+=`, celles des autres s'en déduit immédiatement.

- Syntaxe :

  expression :

  ⇒ *lvalue* `+=` *expression*

- Sémantique :

  lvalue `=` *lvalue* `+` *expression*

7.3 Opérateur *conversion*

- Syntaxe :

  expression :

  ⇒ `(` *type* `)` *expression*

- Sémantique : *expression* est évaluée et convertie dans le type indiqué par *type*.

Note

Dans le jargon C, l'opérateur de conversion de type s'appelle un *cast*. Dans le vocabulaire des langages de programmation en général, une conversion de type s'appelle en anglais une *coertion*, que l'on peut traduire par contrainte ou coercition. Le mot anglais *cast* signifie plâtre (pour maintenir un membre brisé), il donne donc bien une idée de contrainte, mais c'est quand même un choix bizarre.

Exemples d'utilisation

L'opérateur de conversion est devenu moins utile avec la normalisation du langage C. Dans K&R C, il était utilisé essentiellement pour deux raisons :

1. à cause de l'absence de pointeur générique `void *`. En effet, les procédures d'allocation de mémoire comme `malloc` étaient définies par :

   ```
   extern char * malloc();
   ```

   ce qui nécessitait d'utiliser l'opérateur de conversion de type à chaque utilisation :

   ```
   p1 = (struct s1 *) malloc(sizeof(struct s1));
   p2 = (struct s2 *) malloc(sizeof(struct s2));
   ```

2. à cause de l'absence de prototype de fonction qui rendait impossible la déclaration du type des paramètres des fonctions externes. Si une procédure `p` attendait un paramètre de type `float`, et si on désirait lui passer la valeur possédée par la variable `i` de type `int`, il ne fallait pas écrire `p(i)` mais `p((float) i)`.

Il reste cependant un certain nombre de situations où l'opérateur de conversion est nécessaire. En voici un exemple. Il s'agit d'un programme qui a pour but de déterminer si l'architecture de la machine est de type *little endian* ou *big endian*. Il faut regarder l'ordre des octets dans un entier, d'où la nécessité de l'opérateur de conversion. Ce programme suppose que les *int* sont implémentés sur 4 octets.

```
int i = 0x01020304;
char *p;

p = (char *) &i;    /*   int * transformé en char *   */
if (*p++ == 1 && *p++ == 2 && *p++ == 3 && *p++ == 4 )
   printf("big endian\n");
else
   {
   p = (char *) &i;
   if (*p++ == 4 && *p++ == 3 && *p++ == 2 && *p++ == 1 )
      printf("little endian\n");
   else printf("architecture exotique !!\n");
   }
```

Exécuté sur une machine Sun à processeur SPARC, ce programme répondra *big endian*, exécuté sur un PC à processeur Intel, il répondra *little endian*.

7.4 Sémantique des expressions

7.4.1 Opérateurs d'adressage

Dans le langage C, les constructions suivantes :

`()`	pour l'appel de procédure
`[]`	pour l'indexation
`*`	pour l'indirection
`.`	pour la sélection de champ
`->`	pour l'indirection et sélection
`&`	pour délivrer l'adresse d'un objet

sont des opérateurs à part entière. Cela signifie que ces opérateurs, que l'on peut appeler opérateurs d'adressage, ont une priorité et sont en concurrence avec les autres opérateurs pour déterminer la sémantique d'une expression. Par exemple, la sémantique de l'expression `*p++` ne peut se déterminer que si l'on connaît les priorités relatives des opérateurs `*` et `++`.

7.4.2 Priorité et associativité des opérateurs

Pour déterminer la sémantique d'une expression il faut non seulement connaître la priorité des opérateurs mais également leur associativité. En effet, seule la connaissance de l'associativité de l'opérateur `==` permet de savoir si `a == b == c` signifie `(a == b) == c` ou si elle signifie `a == (b == c)`.

Un opérateur a une associativité à droite quand :
`a` *op* `b` *op* `c` signifie `a` *op* `( b` *op* `c)`.

Un opérateur a une associativité à gauche quand :
`a` *op* `b` *op* `c` signifie `(a` *op* `b)` *op* `c`.

Nous donnons ci-dessous le tableau exhaustif des opérateurs avec leurs priorités et leurs associativité.

priorité	Opérateur	Associativité
16	`()` `[]` `->` `.` `++`[a] `--`[b]	G
15	`!` `~` `++`[c] `--`[d] `-`[e] `+`[f] `*`[g] `&`[h] `sizeof`	D
14	*conversion*	D
13	`*`[i] `/` `%`	G
12	`+` `-`	G
11	`<<` `>>`	G
10	`<` `<=` `>` `>=`	G
9	`==` `!=`	G
8	`&`[j]	G
7	`^`	G
6	`|`	G
5	`&&`	G
4	`||`	G
3	`?:`	D
2	`=` `+=` `-=` `*=` `/=` `%=` `>>=` `<<=` `&=` `^=` `|=`	D
1	`,`	G

[a] postfixé
[b] postfixé
[c] préfixé
[d] préfixé
[e] unaire
[f] unaire
[g] indirection
[h] adresse de
[i] multiplication
[j] et bit bit

Discussion

Les choix faits pour les priorités des opérateurs sont assez mauvais, les concepteurs du langage eux-mêmes en conviennent.[1] Les choix les plus irritants sont les suivants :

- La précédence des opérateurs bits à bits est plus petite que celle des opérateurs de comparaison. Donc `a&b == c` ne signifie pas `(a&b) == c`, mais `a & (b==c)`.

- La précédence des opérateurs de décalage est plus petite que celle des opérateurs de `+` et `-`. Donc `a << 4 + b` signifie `a << (4 + b)`.

Recommandation

Il est considéré comme un bon style de programmation en C, de systématiquement parenthéser les expressions dès qu'elles comportent d'autres opérateurs que les opérateurs de l'arithmétique usuelle.

1. The C programming langage, page 3 : C, like any other language, has its blemishes. Some of the operators have the wrong precedence;

7.4.3 Ordre d'évaluation des opérandes

À part quelques exceptions, l'ordre d'évaluation des opérandes d'un opérateur n'est pas spécifié par le langage. Ceci a pour conséquence que le programmeur doit faire extrêmement attention aux effets de bords dans les expressions. Par exemple, l'instruction :

```
t[i] = f();
```

où la fonction `f` modifie la valeur de `i` a un comportement indéterminé : il est impossible de savoir si la valeur prise pour indexer `t` sera celle de `i` avant ou après l'appel à `f`.

7.5 Récréation

Voici en illustration de l'opérateur `~`, la contribution de Jack Applin a la compétition du code C le plus obscur (IOCCC) de 1986. Ce programme a la propriété extraordinaire d'être un source valide à la fois pour le shell `/bin/sh`, le langage C et FORTRAN ! Voici le source :

```
cat =13 /*/ >/dev/null 2>&1; echo "Hello, world!"; exit
*
*  This program works under cc, f77, and /bin/sh.
*
*/; main() {
      write(
cat-~-cat
     /*,'(
*/
     ,"Hello, world!"
     ,
cat); putchar(~-~-~-cat); } /*
     ,)')
      end
*/
```

La version shell

La commande `cat =13` est une commande incorrecte (à cause du blanc entre `cat` et le signe `=`), mais comme l'*erreur standard* est redirigée sur `/dev/null`, le message d'erreur n'apparaît pas. Ensuite, l'echo imprime « Hello world », puis le shell fait exit. Le reste du source lui est donc indifférent. Ensuite, les deux versions C et fortran sont mélangées grâce à un emploi judicieux des commentaires (en fortran, toute ligne commençant par `*` ou `c` est un commentaire).

La version fortran

Une fois débarrassé des commentaires fortran, le source devient :

```
      write( *,'("Hello, world!")')
      end
```

ce qui imprime « Hello world ».

La version C

Après nettoyage des commentaires C, le source original devient :

```
cat =13 ;
main()
{
write( cat-~-cat ,"Hello, world!" , cat);
putchar(~-~-~-cat);
}
```

La déclaration `cat =13 ;` est valide en C K&R mais obsolète en ANSI C : elle est équivalente à `int cat =13 ;` Cette forme est cependant encore généralement acceptée (avec un warning) par les compilateurs. La suite ne fonctionne correctement que sur une machine satisfaisant aux deux contraintes suivantes :

1. être une machine UNIX pour disposer de l'appel noyau `write` ;
2. avoir les entiers négatifs représentés en complément à 2. Dans ce cas en effet, `~-x` vaut `x - 1`.

Donc `cat-~-cat` vaut 1 qui, en premier paramètre de `write` désigne la *sortie standard*, et `~-~-~-13` vaut 10 (le code de *newline*). Le troisième paramètre passé à `write` doit être la longueur de la chaîne à imprimer, ici 13 qui est bien la longueur de `Hello, world!`. Au final, ce programme imprime « Hello world ».

Chapitre 8

Le préprocesseur

Les services rendus par le préprocesseur sont : l'inclusion de fichier source, le traitement de macros et la compilation conditionnelle. L'inclusion de fichier source a déjà été vue dans le chapitre 1.14 nous n'y reviendrons pas.

8.1 Traitement de macros

Il existe deux types de macros : les macros sans paramètre et les macros avec paramètres.

8.1.1 Les macros sans paramètres

Les macros sans paramètre ont été introduites au paragraphe 1.10.1. Rappelons que lorsque le préprocesseur lit une ligne du type :

`#define` *nom* *reste-de-la-ligne*

il remplace dans toute la suite du source, toute nouvelle occurrence de *nom* par *reste-de-la-ligne*. Il n'y a aucune contrainte quand à ce qui peut se trouver dans *reste-de-la-ligne*, mais l'utilité principale des macros sans paramètre est de donner un nom parlant à une constante. Les avantages à toujours donner un nom aux constantes sont les suivants :

1. un nom bien choisi permet d'expliciter la sémantique de la constante. Exemple : `#define NB_COLONNES 100`.

2. la constante chiffrée se trouve à un seul endroit, ce qui facilite la modification du programme quand on veut changer la valeur de la constante (cas de la taille d'un tableau, par exemple).

3. on peut expliciter les relations entre constantes. Exemple :
   ```
   #define NB_LIGNES 24
   #define NB_COLONNES 80
   #define TAILLE_TAB     NB_LIGNES * NB_COLONNES
   ```

Exemple de mauvaise utilisation

Du fait de l'absence de contrainte sur *reste-de-la-ligne*, on peut faire des choses très déraisonnables avec les macros. Un exemple célèbre est le source du shell écrit par Steve Bourne pour le système UNIX. Bourne avait utilisé les facilités de macros pour programmer dans un dialecte de Algol 68. Voici un extrait de ses définitions :

```
#define IF      if(
#define THEN    ){
#define ELSE    } else {
#define ELIF    } else if (
#define FI      ;}

#define BEGIN   {
#define END     }
#define SWITCH  switch(
#define IN      ){
#define ENDSW   }
#define FOR     for(
#define WHILE   while(
#define DO      ){
#define OD      ;}
#define REP     do{
#define PER     }while(
#undef DONE
#define DONE    );
#define LOOP    for(;;){
#define POOL    }
```

Et voici un exemple de code :

```
assign(n,v)
        NAMPTR          n;
        STRING          v;
{
        IF n->namflg&N_RDONLY
        THEN    failed(n->namid,wtfailed);
        ELSE    replace(&n->namval,v);
        FI
}
```

Ce n'est ni du C ni de l'Algol, il y a un consensus dans la communauté C pour estimer que ce genre de choses est à proscrire.

Définition de macro à l'invocation du compilateur

Certains compilateurs permettent de définir des macros sans paramètres à l'invocation du compilateur. Il est alors possible d'écrire un programme utilisant une macro qui n'est nulle part définie dans le source. La définition se fera à l'invocation du compilateur. Ceci est très pratique pour que certaines constantes critiques d'un programme aient une valeur qui soit attribuée à l'extérieur du programme, par une phase de configuration par exemple.

Ci-dessous, un exemple pour le système UNIX : la compilation du fichier `fic.c` en définissant la macro sans paramètre de nom `NB_LIGNES` et de valeur 24 :

```
cc -c -DNB_LIGNES=24 fic.c
```

8.1.2 Macros prédéfinies

Il y a un certain nombre de macros prédéfinies par le préprocesseur :

nom	valeur de la macro	forme syntaxique
`__LINE__`	numéro de la ligne courante du programme source	entier
`__FILE__`	nom du fichier source en cours de compilation	chaîne
`__DATE__`	la date de la compilation	chaîne
`__TIME__`	l'heure de la compilation	chaîne
`__STDC__`	1 si le compilateur est ISO, 0 sinon	entier

8.1.3 Les macros avec paramètres

Une macro avec paramètres se définit de la manière suivante :
`#define` *nom* (*liste-de-paramètres-formels*) *reste-de-la-ligne*

La *liste-de-paramètres-formels* est une liste d'identificateurs séparés par des virgules. Le *reste-de-la-ligne* est appelé « corps de la macro ». Toute occurrence ultérieure de *nom* sera un appel de la macro et devra avoir la forme :
nom (*liste-de-paramètres-effectifs*)
Dans la *liste-de-paramètres-effectifs*, les paramètres sont séparés par des virgules et chaque paramètre est une suite quelconque d'unités lexicales. Le préprocesseur remplace l'ensemble nom de la macro et liste de paramètres effectifs parenthésés, par *reste-de-la-ligne* dans lequel chaque paramètre formel est remplacé par le paramètre effectif correspondant. Cette opération de remplacement de texte porte le nom d'*expansion* de la macro.

L'utilité principale des macros avec paramètres est de bénéficier de la clarté d'expression des fonctions sans en souffrir la lourdeur : le code est inséré en ligne, donc on économise le code d'entrée et de retour de fonction. Exemple :

```
#define min(a, b)       ((a) < (b) ? (a) : (b))
#define max(a, b)       ((a) < (b) ? (b) : (a))

f()
{
int i,j,k;

i = min(j,k); /*  équivalent à : i =  j  <  k  ?  j  :  k ;            */
i = max(j,k); /*  équivalent à : i =  j  <  k  ?  k  :  j ;            */
}
```

Attention

La distinction entre macro avec et sans paramètre se fait sur le caractère qui suit immédiatement le *nom* de la macro : si ce caractère est une parenthèse ouvrante c'est une macro avec paramètres, sinon c'est une macro sans paramètre. En particulier, si après le *nom* de la macro il y a un blanc avant la parenthèse ouvrante, ça sera une macro sans paramètre. Exemple :

```
#define CARRE (a) a * a
```

Une utilisation de `CARRE(2)` aura comme expansion `(a) a * a(2)` ! Attention donc à l'erreur difficile à voir : la présence d'un blanc entre le nom d'une macro avec paramètres et la parenthèse ouvrante.

Exemple

Cet exemple est tiré du source de LINUX. Il s'agit d'un fragment de gestion de la mémoire virtuelle, une structure page a été définie :

```
struct page {
   struct inode *inode;
   unsigned long offset;
   struct page *next_hash;
   atomic_t count;
   unsigned flags; /* atomic flags, some possibly updated asynchronously */
   ...    /*   d'autres champs   */
   };
```

Dans cette structure, le champs flags est un ensemble de bits définis ci-après :

```
/* Page flag bit values */
#define PG_locked              0
#define PG_error               1
#define PG_referenced          2
#define PG_uptodate            3
#define PG_free_after          4
#define PG_decr_after          5
```

Puis le programmeur a défini des macros pour tester commodément ces bits à l'aide de la fonction `test_bit` définie par ailleurs :

```
/* Make it prettier to test the above... */
#define PageLocked(page)        (test_bit(PG_locked, &(page)->flags))
#define PageError(page)         (test_bit(PG_error, &(page)->flags))
#define PageReferenced(page)    (test_bit(PG_referenced, &(page)->flags))
#define PageUptodate(page)      (test_bit(PG_uptodate, &(page)->flags))
#define PageFreeAfter(page)     (test_bit(PG_free_after, &(page)->flags))
#define PageDecrAfter(page)     (test_bit(PG_decr_after, &(page)->flags))
```

Exemple de mauvaise utilisation

Dans une invocation de macro, chaque paramètre effectif peut être une suite quelconque d'unités lexicales, mais après expansion, le texte obtenu doit être un fragment valide de langage C. Voici un exemple des horreurs que l'on peut écrire en utilisant les macros :

```
#define macro(a,b) a [ b

f()
{
int i, t[10];

macro(t,i]) = 1;   /*   équivalent à t[i] = 1;   */
}
```

Le second paramètre passé à la macro (`i]`) ne correspond syntaxiquement à rien, mais le résultat de l'expansion de la macro est correct.

8.1.4 Les pièges des macros

Par le fait que le traitement des macros consiste à faire de la substitution de texte, l'écriture de macros recèle de nombreux pièges.

Pièges des priorités d'opérateurs

Supposons que l'on écrive :

```
#define CARRE(a) a * a
```

une occurrence de `CARRE(a+b)` aura comme expansion `a+b * a+b` ce qui est différent du `(a+b) * (a+b)` qui était désiré. De la même manière `!CARRE(x)` aura comme expansion `!x * x` ce qui est différent du `!(x * x)` qui était désiré.

On recommande donc de toujours respecter deux règles dans la définition d'une macro devant être utilisée dans des expressions :

1. parenthéser les occurrences des paramètres formels ;
2. parenthéser le corps complet de la macro.

Une définition de `CARRE` respectant ces règles est :

```
#define CARRE(a) ((a) * (a))
```

Pièges des effets de bord

L'utilisation d'effet de bord sur les paramètres effectifs d'une macro peut avoir des effets complètement inattendus. Après la définition :

```
#define CARRE(a) ((a) * (a))
```

l'utilisation de `CARRE(x++)` aura comme expansion `((x++) * (x++))`, l'opérateur `++` sera donc appliqué deux fois.

8.1.5 Macros générant des instructions

Tous les exemples donnés jusqu'ici sont des exemples de macros générant des expressions. Les macros peuvent aussi générer des instructions et là aussi il y a des pièges à éviter. Supposons qu'ayant à écrire un grand nombre de fois un appel de fonction avec test d'erreur, on définisse la macro suivante :

```
#define F(x) if (!f(x)) { printf("erreur\n"); exit(1); }
```

La macro pourra s'appeler comme une fonction (avec un `;` à la fin) dans un contexte de liste d'instructions :

```
{
...
F(i);
...
}
```

Par contre, dans un contexte d'instruction (et non de liste d'instructions), il ne faudra pas mettre de `;` à la fin :

```
do  F(a)  while ( ... );
```

alors qu'il le faudrait si il s'agissait d'une fonction :

```
do  f(a);  while ( ... );
```

Mais le pire reste à venir : voyons ce qui se passe si on utilise la macro `F` dans un `if` avec `else` :

```
if ( ... )
   F(i)
else
   ...
```

Il suffit d'imaginer l'expansion de la macro :

```
if ( ... )
   if (!f(x)) { printf("erreur\n"); exit(1); }
else
   ...
```

pour comprendre le problème : le `else` va être raccroché au `if` de la macro, ce qui n'est pas ce qu'a voulu le programmeur.

Recommandation

Pour toute macro générant des instructions, on recommande d'englober les instructions générées dans la partie instruction d'un `do ... while (0)`. Notre exemple s'écrit ainsi :

```
#define F(x) do if (!f(x)) { printf("erreur\n"); exit(1); } while (0)
```

et tous les problèmes précédents s'évanouissent.

8.2 Compilation conditionnelle

Les mécanismes de compilation conditionnelles ont pour but de compiler ou d'ignorer des ensembles de lignes, le choix étant basé sur un test exécuté à la compilation.

8.2.1 Commande `#if`

La commande permettant de réaliser la compilation conditionnelle est la commande `#if` qui peut prendre plusieurs formes.

Commande `#if` simple

Quand le préprocesseur rencontre :
`#if` *expression*
ensemble-de-lignes
`#endif`
il évalue *expression*. Si *expression* délivre une valeur non nulle, *ensemble-de-lignes* est compilé, sinon *ensemble-de-lignes* est ignoré. L'évaluation de *expression* a lieu au moment de la compilation, elle ne doit donc comporter que des constantes. L'*ensemble-de-lignes* est une suite de lignes quelconques.

Commande `#if` avec `#else`

Sur rencontre de :
`#if` *expression*
ensemble-de-lignes$_1$
`#else`
ensemble-de-lignes$_2$
`#endif`
le préprocesseur évalue *expression*. Si *expression* délivre une valeur non nulle, *ensemble-de-lignes*$_1$ est compilé et *ensemble-de-lignes*$_2$ est ignoré, sinon *ensemble-de-lignes*$_1$ est ignoré et *ensemble-de-lignes*$_2$ est compilé.

Commande `#if` avec `#elif`

De manière à imbriquer aisément des `#if` dans des `#if`, il existe une commande `#elif` dont la sémantique est else if. Elle s'utilise de la manière suivante :
`#if` *expression*$_1$
ensemble-de-lignes$_1$
`#elif` *expression*$_2$
ensemble-de-lignes$_2$
...
`#elif` *expression*$_n$
ensemble-de-lignes$_n$
`#else`
ensemble-de-lignes$_{else}$
`#endif`
Un seul *ensemble-de-lignes* sera compilé : celui correspondant à la première *expression*$_i$ qui

s'évaluera à une valeur non nulle si elle existe, ou bien *ensemble-de-lignes*$_{else}$ si toutes les *expression*$_i$ s'évaluent à 0.

8.2.2 Commandes `#ifdef` et `#ifndef`

Dans ce qui précède il est possible de remplacer les commandes `#if` *expression* par :
`#ifdef` *nom* ou
`#ifndef` *nom*.
Dans ce cas, le test ne porte plus sur la nullité ou non d'une expression, mais sur la définition ou non d'une macro. La commande `#ifdef` *nom* a pour sémantique : « si *nom* est défini », et `#ifndef` *nom* a pour sémantique : « si *nom* n'est pas défini ».

8.2.3 L'opérateur `defined`

L'opérateur `defined` est un opérateur spécial : il ne peut être utilisé que dans le contexte d'une commande `#if` ou `#elif`. Il peut être utilisé sous l'une des deux formes suivantes : `defined` *nom* ou bien : `defined (` *nom* `)`. Il délivre la valeur 1 si *nom* est une macro définie, et la valeur 0 sinon. L'intérêt de cet opérateur est de permettre d'écrire des tests portant sur la définition de plusieurs macros, alors que `#ifdef` ne peut en tester qu'une.

```
#if defined(SOLARIS) || defined(SYSV)
```

8.2.4 La commande `#error`

La commande `#error` a la syntaxe suivante : `#error` *suite-d-unités-lexicales*
La rencontre de cette commande provoquera l'émission d'un message d'erreur comprenant la *suite-d-unités-lexicales*. Cette commande a pour utilité de capturer à la compilation des conditions qui font que le programme ne peut pas s'exécuter sur cette plate-forme. Voici un exemple où on teste que la taille des entiers est suffisante :

```
#include <limits.h>
#if INT_MAX < 1000000
#error "Entiers trop petits sur cette machine"
#endif
```

8.2.5 Usage

La compilation conditionnelle a pour but essentiel d'adapter le programme à son environnement d'exécution : soit il s'agit d'un programme système devant s'adapter au matériel sur lequel il s'exécute, soit il s'agit d'un programme d'application qui doit s'adapter au système sur lequel il s'exécute. Prenons par exemple le système UNIX qui existe en deux grandes variantes : la variante BSD et la variante SYSTEM V. La routine de recherche d'un caractère déterminé dans une chaîne s'appelle `index` en BSD et `strchr` en SYSTEM V, mais l'interface est le même. Voici comment on peut écrire un programme se compilant et s'exécutant sur les deux plate-formes : le programmeur peut décider d'utiliser un nom à lui, par exemple RechercheCar, et de le définir comme ci-dessous.

```
#if defined(HAS_INDEX)
```

```
#define RechercheCar index
#elif defined(HAS_STRCHR)
#define RechercheCar strchr
#else
#error "Impossible de réaliser RechercheCar"
#endif
```

Selon le système, la compilation se fera par :

```
cc -c -DHAS_INDEX fichier.c
```

ou par :

```
cc -c -DHAS_STRCHR fichier.c
```

8.3 Récréation

Quel est le plus petit programme possible en C ?

Mark Biggar a été un vainqueur de la compétition du code C le plus obscur (IOCCC) avec un programme ne comportant qu'une seule lettre : `P` ! Pour arriver à ce résultat, il avait compliqué un petit peu la ligne de commande de compilation :

```
cc -DC="R>0" -DI="if(T)O" -DO="c=write(1,&c,1);" -DP="main(){X}" \
-DR="read(0,&c,1)" -DT="c!=015" -DW="while(C)I" -DX="char c;W" markb.c
```

Le fichier `markb.c` contenant la lettre `P`, qui du fait de l'expansion des macros, va être transformée en :

```
main()
{
char c;
while(read(0,&c,1) >0)
   if (c!=015) c=write(1,&c,1);
}
```

Chapitre 9

Les déclarations

Nous n'avons vu jusqu'à présent que des exemples de déclarations, il est temps maintenant de voir les déclarations de manière plus formelle.

9.1 Déclarations de définition et de référence

Il existe dans le langage C, deux types de déclarations :

– les déclarations qui définissent complètement un objet, ce sont les déclarations de définition ;

– les déclarations qui font référence à un objet défini ailleurs, ce sont les déclarations de référence.

Les déclarations de définition

Ce sont celles que l'on utilise dans la grande majorité des cas : on définit un objet et ensuite on l'utilise.

Les déclarations de référence

Elles sont nécessaires pour :

– un nom de variable ou de fonction défini dans une autre unité de compilation.

– un nom de fonction défini dans la même unité de compilation, pour résoudre le cas d'appel récursif : la fonction `f1` appelle `f2` qui appelle `f3`, ... qui appelle `fn` qui appelle `f1`.

– un nom de structure ou d'union défini dans la même unité de compilation, pour résoudre le cas de référence récursive : la `struct s1` possède un champ dont le type référence la `struct s2` qui possède un champ dont le type référence la `struct s3` qui etc. jusqu'à revenir à la `struct s1`.

9.1.1 Déclarations de variables

Différences entre déclarations et définitions de variables :

- une déclaration de référence est précédée du mot-clé `extern` ;
- une déclaration de référence peut avoir un type incomplet : absence de la taille d'un tableau.

Exemples :

```
int i;                  /*   définition de i                            */
extern int j;           /*   référence à un entier défini ailleurs      */
int t1[20];             /*   définition de t                            */
extern t2[];            /*   référence à un tableau t2 défini ailleurs  */
```

9.1.2 Déclarations de fonctions

Une déclaration de fonction ayant la partie instruction est une définition, une déclaration de fonction n'ayant pas de partie instruction est une déclaration de référence. La présence ou l'absence du mot-clé `extern` dans l'un ou l'autre cas est possible, mais on considère comme un bon style de programmation de le mettre à une déclaration de référence et de l'omettre à une définition. Exemples :

```
/* max n'a pas de partie instruction : déclaration de référence        */
extern int max(int a, int b) ;

/*   min possède la partie instruction : c'est une définition          */
int min(int a, int b)
{
return(a < b ? a : b);
}
```

9.1.3 Déclarations d'étiquettes de structures et union

Attention : il ne s'agit pas de déclaration de noms de variables de type structure ou union, mais de déclaration d'étiquette de structure ou union. Quand on veut déclarer une étiquette de structure ou union qui sera définie plus tard, on procède de la manière suivante :

```
struct str1;           /*   déclaration de référence de str1       */
struct str2            /*   définition de str2 qui référence str1   */
   {
   ...
   struct str1 * p;
   };

struct str1            /*   définition de str1 qui référence str2   */
   {
   ...
   struct str2 *p;
   };
```

Le mécanisme est le même avec les unions.

9.2 Portée des déclarations

Il existe en C quatre types de portées possibles pour les déclarations :

- un identificateur déclaré à l'extérieur de toute fonction, a une portée qui s'étend de son point de déclaration jusqu'à la fin du source ;
- un paramètre formel de fonction a une portée qui s'étend de son point de déclaration jusqu'à la fin de l'instruction composée formant le corps de la fonction ;
- un identificateur déclaré dans une instruction composée a une portée qui s'étend du point de déclaration jusqu'à la fin de l'instruction composée ;
- une étiquette d'instruction a une portée qui comprend tout le corps de la fonction dans laquelle elle apparaît.

Exemple :

```
int i;              /*   déclaration à l'extérieur de toute fonction     */

void proc1(int j)   /*  j paramètre de la procédure proc1                */
{

...                 /*   instructions 1                                  */

k:
if (...)
   {
   int l;       /*  déclaration à l'intérieur d'une instruction composée  */

   ...          /*   instructions 2                                      */
   }
}               /*   fin de proc1                                        */

int func1()     /*   début de func1                                      */
{
...             /*   instructions 3                                      */
}               /*   fin de func1                                        */
```

Dans cet exemple,
`i` pourra être référencé par $instructions_1$, $instructions_2$ et $instructions_3$,
`j` pourra être référencé par $instructions_1$ et $instructions_2$,
`k` pourra être référencé par $instructions_1$ et $instructions_2$,
`l` pourra être référencé par $instructions_2$.

9.3 Visibilité des identificateurs

Dans le langage C, l'imbrication des instructions composées forme une structure classique de blocs, c'est à dire que les déclarations d'une instruction composée englobée cachent les déclarations des instructions composées englobantes ayant le même nom. De surcroît, les déclarations d'une instruction composée cachent les déclarations de même nom, qui sont à l'extérieur de toute fonction. Exemple :

```
int i;
int j;

void proc1()
{
int i;      /*   cache le i précédent   */
int k;

if (a > b)
   {
   int i;   /*   cache le i précédent   */
   int j;   /*   cache le j précédent   */
   int k;   /*   cache le k précédent   */

   ...
   }
}
```

9.4 Les espaces de noms

9.4.1 Position du problème

Il existe certaines situations où l'on peut accepter que le même nom désigne plusieurs objets différents, parce que le contexte d'utilisation du nom permet de déterminer quel est l'objet référencé. Considérons l'exemple suivant :

```
struct st1
   {
   int i;
   int j;
   };

struct st2
   {
   int i;
   double d;
   };

int main()
{
```

```
struct st1 s1;   /*   déclaration de la variable s1   */
struct st2 s2;   /*   déclaration de la variable s2   */

s1.i = s2.i;
}
```

Dans l'instruction `s1.i = s2.i`, il y a deux occurrence du nom `i`, la première désigne le `i` de `s1`, et la seconde désigne le `i` de `s2`. On voit que le contexte d'utilisation de `i` a permis de déterminer à chaque fois de quel `i` il s'agit. On dit alors que le `i` de `s1` et le `i` de `s2` appartiennent à des *espaces de noms* différents.

9.4.2 Les espaces de noms du langage C

Il y a quatre types d'espaces de noms dans le langage :

- un espace pour les étiquettes de structures, d'unions et d'énumération ;
- un espace pour les noms de champs de structures ou unions : il y a un espace de nom pour **chaque** structure et **chaque** union ;
- un espace pour les étiquettes de branchement ;
- le dernier espace est formé de tous les autres noms.

Nous donnons ci-dessous un exemple où le même identificateur `i` est utilisé de manière valide dans 5 espaces de noms différents.

```
int i;          /*   i est un nom d'identificateur      */

struct i        /*   i est une étiquette de structure   */
   {
   int i;       /*   i est un champ de la struct i      */
   int j;
   }i1,i2;

struct ii
   {
   int i;       /*   i est un champ de la struct ii     */
   int j;
   }ii1,ii2;

int main()
{
i: /*   i est une étiquette de branchement */
i = 1;
i1.i = 2;
ii1.i = 3;
goto i;
}
```

Remarque

Certains auteurs considèrent également qu'il existe un espace de noms pour les noms définis à l'aide de la commande `#define` du macro-processeur. Nous avons refusé cette vision des choses dans la mesure où pendant la phase de compilation proprement dite, ces noms n'ont plus d'existence.

9.5 Durée de vie

Du point de vue de la durée de vie, il existe trois types de variables : les variables *statiques*, les variables *automatiques* et les variables *dynamiques*.

- Les variables *statiques* sont allouées au début de l'exécution du programme, et ne sont libérées qu'à la fin de l'exécution du programme.
- Les variables *automatiques* sont allouées à l'entrée d'une instruction composée, et libérées lors de la sortie de l'instruction composée.
- Les variables *dynamiques* sont allouées et libérées explicitement par le programmeur, à l'aide des fonctions `malloc` et `free`.

Dans les langages de programmation, il y a généralement un lien étroit entre la portée de la déclaration d'une variable et sa durée de vie. Il est classique en effet, qu'une variable globale (c'est à dire dont la déclaration se trouve à l'extérieur de toute fonction), soit une variable *statique*, et qu'une variable locale à une procédure ou fonction, soit une variable *automatique*.

Dans le langage C, le programmeur a davantage de liberté. Les variables globales sont ici aussi des variables statiques, mais les variables locales peuvent être au choix du programmeur statiques ou automatiques. Si la déclaration d'une variable locale est précédée du mot clé `static`, cette variable sera statique, si elle est précédée du mot-clé `auto`, elle sera automatique, et en l'absence de l'un et l'autre de ces mots-clés, elle sera prise par défaut de type automatique.

Discussion

Cette liberté de donner à une variable locale une durée de vie égale à celle du programme, permet de résoudre des problèmes du type exposé ci-dessous. Imaginons une procédure qui pour une raison quelconque doit connaître combien de fois elle a été appelée. Le programmeur a besoin d'une variable dont la durée de vie est supérieure à celle de la procédure concernée, ce sera donc une variable statique. Cependant cette variable doit être une variable privée de la procédure, (il n'y a aucune raison qu'une autre procédure puisse en modifier la valeur), il faut donc que ce soit une variable locale à la procédure. En C, on programmera de la manière suivante :

```
void proc()
{
static int nb_appel = 0;

nb_appel++;
...
}
```

Dans d'autres langages, de manière à satisfaire les contraintes de durée de vie, on aurait été obligé de faire de la variable `nb_appel`, une variable globale, la rendant ainsi accessible aux autres procédures, ce qui est tout à fait illogique.

9.6 Classes de mémoire

9.6.1 Position du problème

Lors de l'exécution d'un programme C il y a trois zones de mémoire différentes, correspondant aux trois durées de vies possibles :

- la zone contenant les variables statiques ;
- la zone contenant les variables automatiques (cette zone est gérée en pile puisqu'en C, toute fonction peut être récursive) ;
- la zone contenant les variables dynamiques (cette zone est généralement appelée *le tas*).

Un tel découpage se rencontre couramment dans les langages de programmation. Généralement cependant, il n'est pas nécessaire au programmeur de déclarer la classe dans laquelle il désire mettre une variable, cette classe étant choisie de manière autoritaire par le langage.

Les concepteurs du langage C ont voulu offrir plus de souplesse aux programmeurs. En effet, nous venons de voir que l'on pouvait mettre dans la classe des variables statiques une variable qui était locale à une instruction composée. D'autre part, pour des raisons d'efficacité des programmes générés, les concepteurs du langage ont créé une nouvelle classe : la classe `register`. Quand le programmeur déclare par exemple :

```
register int i;
```

ceci est une indication au compilateur, lui permettant d'allouer la variable dans une ressource de la machine dont l'accès sera plus rapide que l'accès à une mémoire (si une telle ressource existe).

9.6.2 Les spécificateurs de classe de mémoire

Il existe 5 mots-clés du langage que la grammaire nomme *spécificateur de classe de mémoire*. Il s'agit des mots-clés suivants :

auto Ce spécificateur de classe mémoire n'est autorisé que pour les variables locales à une instruction composée. Il indique que la variable concernée à une durée de vie locale à l'instruction composée. Si la déclaration d'une variable locale ne comporte pas de spécificateurs de classe de mémoire, c'est `auto` qui est pris par défaut. Exemple :

```
{
auto int i;
...
}
```

static Ce spécificateur de classe mémoire est autorisé pour les déclarations de variables et de fonctions. Pour les déclarations de variables, il indique que la variable concernée a une durée de vie globale. Dans tous les cas, (variables et fonctions), il indique que le nom concerné ne **doit pas** être exporté par l'éditeur de liens. Exemple :

```
static int i;     /*   i ne sera pas exporté par l'éditeur de liens   */
int j;            /*   j sera exporté par l'éditeur de liens          */

static void f()   /*   f ne sera pas exporté par l'éditeur de liens   */
{
static int k;     /*   k aura une durée de vie globale                */
...
}

void g()          /*   g sera exportée par l'éditeur de liens         */
{
...
}
```

register Ce spécificateur n'est autorisé que pour les déclarations de variables locales à une instruction composée, et pour les déclarations de paramètres de fonctions. Sa signification est celle de `auto` avec en plus une indication pour le compilateur d'allouer pour la variable une ressource à accès rapide. Le programmeur est supposé mettre une variable dans la classe `register` quand elle est fortement utilisée par l'algorithme. Il y a cependant une contrainte : une telle variable n'a pas d'adresse, impossible donc de lui appliquer l'opérateur `&`.

extern Ce spécificateur est autorisé pour les déclarations de variables et de fonctions. Il sert a indiquer que l'objet concerné a une durée de vie globale et que son nom est connu de l'éditeur de liens.

typedef Ce spécificateur n'a rien à voir avec les classes de mémoire : il sert à définir des types. Son utilité sera vue plus loin.

Discussion

On peut faire les critiques suivantes :

1. `auto` ne peut servir que pour les variables locales, mais si on ne le met pas, il est pris par défaut. Conclusion : il ne sert à rien.

2. `static` sert à deux choses très différentes : il permet de rendre statique une variable locale, et c'est une utilisation légitime, mais il sert aussi à cacher à l'éditeur de liens les variables globales. On rappelle que par défaut (c'est à dire sans le mot-clé `static`,) **les variables globales sont connues de l'éditeur de liens**. Il aurait mieux valu faire l'inverse : que par défaut les variables globales soient cachées à l'éditeur de liens, et avoir un mot-clé (par exemple `export`), pour les lui faire connaître.

3. `register` a été introduit dans le langage pour optimiser les programmes. Avec les techniques modernes de compilation, il ne sert à rien car le compilateur est mieux à même que le programmeur d'allouer les registres de la machine de manière efficace. Ne pas oublier en outre, qu'il y a une contrainte attachée à l'utilisation de variables `register` : l'impossibilité de leur appliquer l'opérateur `&`.

4. `extern` et `typedef` n'ont rien à voir avec les classes de mémoire : ils ne sont là que pour des raisons syntaxiques.

9.7 La compilation séparée

9.7.1 Généralités

Dès que l'on programme une application un peu conséquente, il devient nécessaire pour des raisons pratiques d'en fractionner le source en plusieurs unités de compilation. Chaque unité est compilée séparément, puis les binaires obtenus sont liés à l'aide d'un éditeur de liens pour créer le programme désiré. Pour permettre cela, le langage doit disposer d'un moyen d'exprimer que certaines variables ou procédures sont partagées par plusieurs unités de compilation. La méthode la plus répandue pour résoudre ce problème est la méthode dite « des réfs et des défs ». Pour le seul cas des variables, il existe une autre méthode dite « du *common* ».

Méthode des réfs et des défs

Dans cette méthode les déclarations sont dissymétriques : certaines déclarations sont des définitions, et d'autres déclarations sont des références. Une seule unité de compilation doit définir un nom, et plusieurs unités de compilation peuvent le référencer.

Méthode du *common*

Dans cette méthode, les variables partagées sont déclarées comme appartenant à un segment spécial, appelé *common*. Toute variable du *common* est référençable par n'importe quelle unité de compilation. C'est la méthode utilisée par FORTRAN.

9.7.2 La méthode du langage C

C'est un joyeux mélange des deux ! En effet, le langage permet de définir, référencer et mettre dans le *common*. La distinction entre ces trois cas se fait de la manière suivante :

Si la déclaration	Il s'agit d'une
comporte un initialisateur (avec ou sans mot-clé `extern`)	définition
comporte le mot-clé `extern` mais pas d'initialisateur	référence
ne comporte ni initialisateur ni le mot-clé `extern`	mise dans le *common*

Exemples :

La déclaration	est une
`extern int i = 1;`	définition
`int i = 1;`	définition
`extern int i;`	référence
`int i;`	mise dans le *common*

Les contraintes

Au moment de l'édition de liens, pour un identificateur donné, n unités de compilation l'auront défini, p l'auront référencé, et q auront demandé une mise dans le *common*. Les contraintes sont les suivantes :

- n doit valoir 0 ou 1 : au plus une unité de compilation doit définir un nom ;
- si n vaut 0, alors q ne peut être nul : il ne peut pas y avoir que des références.

Voyons les choses sous l'angle de ce qui est autorisé. On peut avoir :

- une définition avec p (évent. 0) références, et q (évent. 0) demandes de mise dans le *common* ;
- p (évent. 0) références et q (non 0) demandes de mise dans le *common*.

Ceci fait beaucoup de possibilités dont deux seulement sont « raisonnables » :

- une définition avec que des références : on adhère strictement à la méthode des réfs et des défs ;
- que des demandes de mise dans le common : on adhère strictement à la méthode du *common*.

Ces deux méthodes sont résumées dans le tableau suivant :

méthode	unité de compilation 1	unité de compilation 2	...	unité de compilation n
réfs défs	`int i = 0;`	`extern int i;`	...	`extern int i`
common	`int i ;`	`int i ;`	...	`int i ;`

En pratique

Les bons auteurs recommandent de s'en tenir à la stricte méthode des réfs et défs, mais la lecture de nombreux sources montre que c'est la méthode du *common* qui a la faveur des programmeurs. Cette méthode à en effet un avantage pratique : dans toutes les unités de compilations, la déclaration d'un nom est strictement la même. Cette déclaration pourra donc être mise dans un fichier qui sera inclus (par `#include`) dans toutes les unités de compilation qui utilisent ce nom.

La méthode des réfs et des défs par contre, impose d'avoir au moins deux déclarations différentes pour un même nom : une pour la définition et une autre (qui peut être dans un fichier d'inclusion) pour la référence.

Le cas des fonctions

Le cas des fonctions est plus simple dans la mesure où il n'y a pas de mise dans le *common*. Les fonctions adhèrent donc strictement à la méthode des réfs et des défs, avec une distinction facile à faire : comme on l'a vu en 9.1.2, si une déclaration de fonction comporte une partie instruction c'est une définition, sinon c'est une référence.

9.8 Définition de types

Il existe en C un moyen de donner un nom à un type. Il consiste à faire suivre le mot clé `typedef` d'une construction ayant exactement la même syntaxe qu'une déclaration de variable. L'identificateur qui est le nom de la variable dans le cas d'une déclaration de variable, est le nom du type dans le cas d'un `typedef`. Exemple :

```
typedef int tab[10];
```

déclare `tab` comme étant le type tableau de 10 entiers, et :

```
typedef struct
   {
   char nom[20];
   int no_ss;
   } personne;
```

déclare `personne` comme étant le type structure à deux champs : un tableau de 20 caractères et un `int`. Ces noms de type sont ensuite utilisables dans les déclarations de variables, exactement comme un type de base :

```
tab t1,t2;          /*   t1 et t2 tableaux de 10 entiers      */
personne *p1,*p2;   /*   p1 et p2 pointeurs vers des struct   */
```

9.9 Utilité des `typedef`

La principale utilité des `typedef` est, si l'on en fait une utilisation judicieuse, de faciliter l'écriture des programmes, et d'en augmenter la lisibilité.

9.9.1 Restriction d'un type de base

Il est parfois nécessaire de manipuler des variables qui ne peuvent prendre comme valeurs qu'un sous-ensemble des valeurs d'un type de base. Supposons que nous voulions manipuler des booléens. Comme le type booléen n'existe pas dans le langage, il faudra utiliser des `int`, en se restreignant à deux valeurs, par exemple 0 et 1. Il est alors intéressant de redéfinir à l'aide d'un `typedef`, le type `int`. On écrira par exemple :

```
#define VRAI 1
#define FAUX 0
typedef int BOOLEAN;
```

On pourra par la suite déclarer des « booléens » de la manière suivante :

```
BOOLEAN b1,b2;
```

et les utiliser :

```
b1 = VRAI;
if (b2 == FAUX) ...
```

mais bien entendu, ce sera à la charge du programmeur d'assurer que les variables `b1` et `b2` ne prennent comme valeurs que `VRAI` ou `FAUX`. Le compilateur ne protestera pas si on écrit :

```
b1 = 10;
```

On voit que la lisibilité du programme aura été augmentée, dans la mesure où le programmeur aura pu expliciter une restriction sémantique apportée au type `int`.

9.9.2 Définition de type structure

Lorsqu'on donne un nom à un type structure par `typedef`, l'utilisation est beaucoup plus aisée. En effet, si on déclare :

```
struct personne
   {
   ...
   }
```

les déclarations de variables se feront par :

```
struct personne p1,p2;
```

alors que si on déclare :

```
typedef struct
   {
   ...
   } PERSONNE;
```

les déclarations de variables se feront par :

```
PERSONNE p1,p2;
```

on voit que la seconde méthode permet d'éviter d'avoir à répéter `struct`.

De la même manière, en ce qui concerne les pointeurs, il est plus difficile d'écrire et de comprendre :

```
struct personne
   {
   ...
   };
struct personne *p1,*p2;       /*   p1 et p2 pointeurs vers des struct   */
```

que la version suivante qui donne un nom parlant au type pointeur vers struct :

```
typedef struct
   {
   ...
   } PERSONNE;

typedef PERSONNE *P_PERSONNE;   /*   P_PERSONNE type pointeur vers struct */

P_PERSONNE p1,p2;               /*   p1 et p2 pointeurs vers des struct   */
```

9.9.3 Définition de types opaques

Dans le cadre de l'écriture de programme en plusieurs unités de compilation, il est souvent utile de définir un type de manière opaque, c'est à dire d'en laisser libre l'utilisation sans que l'utilisateur n'ait à connaître sa définition. C'est exactement ce que réalise la bibliothèque standard pour le type `FILE` : le programmeur sait que `fopen()` rend une valeur de type `FILE *` et que cette valeur doit être passée en paramètre des fonctions d'entrées-sorties `fprintf()`, `fputc()`, `fputs()` etc. Il y a beaucoup d'autres exemples de ce type dans la bibliothèque standard.

9.10 Qualificatifs de type

Il y a deux façons possibles de qualifier un type : par `const` ou par `volatile`.

Qualificatif const

Une variable dont le type est qualifiée par `const` ne peut pas être modifiée. Le programmeur entend ainsi se protéger contre une erreur de programmation. Ceci n'est utile que pour les paramètres d'une fonction, lorsqu'on désire protéger le paramètres effectifs de la fonction en les mettant en « lecture seulement » pour la fonction.

Qualificatif `volatile`

En qualifiant par `volatile` le type d'une variable, le programmeur prévient le compilateur que cette variable peut être modifiée par un moyen extérieur au programme. Ceci se produit lorsqu'on interagit avec des parties matérielles de la machine : coupleurs d'entrées-sorties généralement. Lorsqu'une variable est de type `volatile` le compilateur ne doit pas procéder aux optimisations qu'il réalise sur les variables normales.

Les qualificatifs de type deviennent pénibles en conjonction avec les pointeurs car on a les trois possibilités :

- l'objet pointé est qualifié ;
- le pointeur lui-même est qualifié ;
- le pointeur et l'objet pointé sont qualifiés.

type	sémantique
`const char c;`	caractère constant
`const char *p;`	pointeur vers caractère constant
`char * const p;`	pointeur constant vers caractère
`const char * const p;`	pointeur constant vers caractère constant

9.11 Fonction à nombre variable de paramètres

Il est possible de déclarer une fonction comme ayant un nombre variable de paramètres en « déclarant » les paramètres optionnels par l'unité lexicale `...` (3 points à la suite). Une fonction peut avoir à la fois des paramètres obligatoires et des paramètres optionnels, les paramètres obligatoires apparaissant en premier et l'unité lexicale `...` apparaissant en dernière position dans la liste de déclaration des paramètres formels.

Dans le corps de la fonction on ne dispose pas de nom pour désigner les paramètres. L'accès à ceux-ci ne peut se faire qu'en utilisant les macros suivantes définies dans la bibliothèque standard :

`va_list` permet de déclarer une variable opaque au programmeur, à passer en paramètre aux autres macros. Cette variable s'appelle traditionnellement `ap` (pour *argument pointer*), et a pour but de repérer le paramètre effectif courant.

`va_start` doit être appelée avant toute utilisation de `va_arg`. La macro `va_start` a deux paramètres : la variable `ap` et le nom du dernier paramètre obligatoire de la fonction.

`va_arg` délivre le paramètre effectif courant : le premier appel à `va_arg` délivre le premier paramètre, puis chaque nouvel appel à `va_arg` délivre le paramètre suivant. La macro `va_arg` admet deux paramètres : la variable `ap` et le type du paramètre courant.

`va_end` doit être appelée après toutes les utilisations de `va_arg`. La macro `va_end` admet un seul paramètre : la variable `ap`.

Rien n'est prévu pour communiquer à la fonction le nombre et le type des paramètres effectivement passés : c'est un problème à la charge du programmeur.

9.11.1 Exemple 1

Ci-dessous l'exemple de la fonction **addn** qui réalise la somme de ses paramètres optionnels.

```
#include <stdio.h>
#include <stdarg.h>
/*****************************************************************************/
/*                                                                           */
/*                              addn                                         */
/*                                                                           */
/*   But:                                                                    */
/*      réalise l'addition d'un nombre variable de paramètres                */
/*                                                                           */
/*****************************************************************************/
int addn(int nbopd, ...)        /*   nbopd = nombre d'opérandes du add      */
{
int i, s = 0;
va_list(ap);                    /*   déclaration de ap                      */

va_start(ap,nbopd);             /*   initialisation de ap                   */
for( i = 1; i <= nbopd; i++)
   s = s + va_arg(ap,int);      /*   va_arg() donne le paramètre courant    */
va_end(ap);                     /*   on a fini                              */
return(s);
}

/*****************************************************************************/
/*                              main                                         */
/*****************************************************************************/
int main()
{
printf("resu = %d\n",add(3,10,11,12));   /*   imprime 33                    */
}
```

Dans cet exemple, le problème du nombre de paramètres effectifs a été réglé par un paramètre obligatoire de la fonction : `nbopd`. En ce qui concerne le type des paramètres effectifs, ils sont tous supposés être entiers, d'où le `va_arg(ap,int)`.

9.11.2 Exemple 2

Dans la bibliothèque standard, il y a deux fonctions utilisées couramment qui admettent un nombre variable de paramètres : ce sont `printf` et `scanf`. Voici leur déclaration dans `stdio.h` :

```
extern int printf(const char *, ...);
extern int scanf(const char *, ...);
```

Ces fonctions doivent connaître le nombre et le type des paramètres optionnels. Ce problème est réglé par les séquences d'échappement se trouvant dans le paramètre obligatoire :

- le nombre de paramètres optionnels est égal au nombre de séquences d'échappement ;
- le type de chaque paramètre optionnel est codé dans sa séquence échappement : `%c` pour *char*, `%d` pour *int*, etc.

9.12 Syntaxe des déclarations

La grammaire des déclarations est la suivante :

déclaration :
  ⇒ *spécificateurs-de-déclaration* *liste-de-déclarateurs-init*$_{option}$ `;`

spécificateurs-de-déclaration :
  ⇒ *spécificateur-de-classe-mémoire* *spécificateurs-de-déclaration*$_{option}$
  ⇒ *spécificateur-de-type* *spécificateurs-de-déclaration*$_{option}$
  ⇒ *qualificatif-de-type* *spécificateurs-de-déclaration*$_{option}$

liste-de-déclarateurs-init :
  ⇒ *déclarateur-init*
  ⇒ *liste-de-déclarateurs-init* `,` *déclarateur-init*

déclarateur-init :
  ⇒ *déclarateur*
  ⇒ *déclarateur* `=` *initialisateur*

spécificateur-de-classe-mémoire :
  ⇒ `auto`
  ⇒ `extern`
  ⇒ `static`
  ⇒ `register`
  ⇒ `typedef`

spécificateur-de-type :
  ⇒ `void`
  ⇒ `char`
  ⇒ `short`
  ⇒ `int`
  ⇒ `long`
  ⇒ `float`
  ⇒ `double`
  ⇒ `signed`
  ⇒ `unsigned`
  ⇒ *spécificateur-de-struct-ou-union*
  ⇒ *spécificateur-d-énumération*
  ⇒ *nom-de-typedef*

spécificateur-de-struct-ou-union :
  ⇒ *struct-ou-union* *identificateur*$_{option}$ `{` *liste-de-déclarations-de-struct* `}`
  ⇒ *struct-ou-union* *identificateur*

struct-ou-union :
  ⇒ `struct`

⇒ `union`

liste-de-déclarations-de-struct :
⇒ *déclaration-de-struct*
⇒ *liste-de-déclarations-de-struct* *déclaration-de-struct*

déclaration-de-struct :
⇒ *liste-de-spécificateurs-et-qualificatifs* *liste-de-déclarateurs-de-struct* ;

liste-de-spécificateurs-et-qualificatifs :
⇒ *spécificateur-de-type* *liste-de-spécificateurs-et-qualificatifs*$_{option}$
⇒ *qualificatif-de-type* *liste-de-spécificateurs-et-qualificatifs*$_{option}$

liste-de-déclarateurs-de-struct :
⇒ *déclarateur-de-struct*
⇒ *liste-de-déclarateurs-de-struct* , *déclarateur-de-struct*

déclarateur-de-struct :
⇒ *déclarateur*
⇒ *déclarateur*$_{option}$: *expression-constante*

spécificateur-d-énumération :
⇒ `enum` *identificateur*$_{option}$ `{` *liste-d-énumérateurs* `}`
⇒ `enum` *identificateur*

liste-d-énumérateurs :
⇒ *énumérateur*
⇒ *liste-d-énumérateurs* , *énumérateur*

énumérateur :
⇒ *identificateur*
⇒ *identificateur* `=` *expression constante*

qualificatif-de-type :
⇒ `const`
⇒ `volatile`

déclarateur :
⇒ *pointeur*$_{option}$ *déclarateur-direct*

déclarateur-direct :
⇒ *identificateur*
⇒ `(` *déclarateur* `)`
⇒ *déclarateur-direct* `[` *expression-constante*$_{option}$ `]`
⇒ *déclarateur-direct* `(` *liste-de-types-de-paramètres* `)`
⇒ *déclarateur-direct* `(` *liste-d-identificateurs*$_{option}$ `)`

pointeur :

$\Rightarrow$ `*` *liste-de-qualificatifs-de-types*$_{option}$

$\Rightarrow$ `*` *liste-de-qualificatifs-de-types*$_{option}$ *pointeur*

liste-de-qualificatifs-de-types :

$\Rightarrow$ *qualificatif-de-type*

$\Rightarrow$ *liste-de-qualificatifs-de-types* *qualificatif-de-type*

liste-de-types-de-paramètres :

$\Rightarrow$ *liste-de-paramètres*

$\Rightarrow$ *liste-de-paramètres* , ...

liste-de-paramètres :

$\Rightarrow$ *déclaration-de-paramètre*

$\Rightarrow$ *liste-de-paramètres* , *déclaration-de-paramètre*

déclaration-de-paramètre :

$\Rightarrow$ *spécificateurs-de-déclaration* *déclarateur*

$\Rightarrow$ *spécificateurs-de-déclaration* *déclarateur-abstrait*$_{option}$

liste-d'identificateurs :

$\Rightarrow$ *identificateur*

$\Rightarrow$ *liste-d'identificateurs* , *identificateur*

initialisateur :

$\Rightarrow$ *expression-d'affectation*

$\Rightarrow$ `{` *liste-d-initialisateurs* `}`

$\Rightarrow$ `{` *liste-d-initialisateurs* , `}`

liste-d'initialisateurs :

$\Rightarrow$ *initialisateur*

$\Rightarrow$ *liste-d'initialisateurs* , *initialisateur*

Exemples

Dans la déclaration :

```
int i,j = 2;
```

`int` est un *spécificateur-de-type* et `i,j = 2` est une *liste-de-déclarateur-init* composée de deux *déclarateur-init* : `i` et `j = 2`. `i` est un *déclarateur-init* sans la partie *initialisateur*, donc réduit à un *déclarateur* lui-même réduit à un *identificateur*. `j = 2` est un *déclarateur-init* comportant un *initialisateur* (`= 2`) et un *déclarateur* réduit à un *identificateur* (`j`). Dans la déclaration :

```
int t[10];
```

`t[10]` est un déclarateur formé d'un *déclarateur* (`t`), suivi de `[` suivi de l'*expression constante* 10, suivi de `]`.

9.13 Sémantique des déclarations

La partie qui nécessite d'être explicitée est la partie de la grammaire concernant les *déclarateur*. La sémantique est la suivante :

- la seule règle de la grammaire qui dérive vers un terminal est la règle :
  déclarateur-direct :
  ⇒ *identificateur*
  ce qui fait qu'a l'intérieur de tout *déclarateur* se trouve un identificateur. Cet identificateur est le nom de l'objet déclaré par la *déclaration*. Exemple :

```
char c;    /*   déclaration de la variable c de type char   */
```

- il y a 3 constructeurs de type :

  1. `*` est un constructeur permettant de construire des types « pointeur vers ... ».
     Exemple :

```
/*   déclaration de p de type pointeur vers short int   */
short int *p;
```

  2. `( )` est un constructeur permettant permettant de construire des types « fonction retournant ... ».
     Exemple :

```
/*  déclaration de sin de type fonction retournant un double  */
double sin();
```

  3. `[` *expression-constante*$_{option}$ `]` est un constructeur permettant de construire des types « tableau de ... ».
     Exemple :

```
/*   déclaration de t de type tableau de 32 int   */
int t[32];
```

- les constructeurs de type peuvent se composer et sont affectés de priorités. Les constructeurs `( )` et `[ ]` ont la même priorité, et celle-ci est supérieure à la priorité du constructeur `*`.

```
char *t[10];        /*   tableau de 10 pointeurs vers des char         */
int *f();           /*   fonction retournant un pointeur vers un int   */
double t[10][10];   /*   tableau de 10 tableaux de 10 double           */
```

- tout comme avec des expressions, la règle :
  déclarateur-direct :
  ⇒ `(` *déclarateur* `)`
  permet de parenthéser des *déclarateur* de manière à en changer la sémantique :

```
char *t[10];     /*   tableau de 10 pointeurs vers un char   */
char (*t)[10];   /*   pointeur vers un tableau de 10 char   */
```

9.14 Discussion sur les déclarations

La syntaxe et la sémantique des déclarations ne sont pas faciles à appréhender dans le langage C pour les raisons que nous allons développer. Tout d'abord, l'ordre des éléments d'une déclaration est assez peu naturel. Au lieu d'avoir comme en PASCAL, une déclaration

```
c : char;
```

qu'on peut traduire par « c est de type char », on a en C :

```
char c;
```

qu'il faut traduire par « de type char est c », ce qui est une inversion peu naturelle. Ensuite, l'identificateur qui est le nom de l'objet déclaré, au lieu d'être mis en évidence dans la déclaration, est caché au beau milieu du *déclarateur*. Exemple :

```
char **p[10];
```

D'autre part, le langage C fait partie des langages qui permettent au programmeur, à partir de types de base et de constructeurs de type, de construire des types complexes. Du point de vue du programmeur, il existe dans le langage les constructeurs suivants :

- `*` pour les pointeurs ;
- `[ ]` pour les tableaux ;
- `( )` pour les fonctions ;
- `struct` pour les structures ;
- `union` pour les unions.

Alors que le programmeur serait en droit de s'attendre à ce que tous ces constructeurs soient traités de manière homogène, en fait, les trois premiers sont des *déclarateur* alors que les deux derniers sont des *spécificateur-de-type*.

Autre point ajoutant encore à la confusion, le constructeur `*` est un constructeur préfixé alors que les constructeurs `[ ]` et `( )` sont postfixé, et le constructeur `*` a une priorité différente de celle des deux autres. Ceci à la conséquence extrêmement désagréable, qu'un type complexe écrit en C ne peut pas se lire de la gauche vers la droite, mais peut nécessiter un analyse du type de celle que l'on fait pour comprendre une expression mathématique.

Par exemple, qui pourrait dire du premier coup d'œil quel est le type de `f` dans la déclaration ci-dessous [1] :

```
char (*(*f())[])()
```

1. `f` est une fonction retournant un pointeur vers un tableau de pointeurs vers une fonction retournant un `char`

9.15 En pratique

Lorsqu'il s'agit d'écrire ou de comprendre un type compliqué, il est recommandé non pas d'utiliser la grammaire des *déclarateur*, mais de partir d'une **utilisation** du type, étant donné que la grammaire est faite de telle sorte que les déclarations calquent très exactement les expressions.

Voyons sur un exemple. Soit à déclarer un pointeur `p` vers une fonction retournant un `int`. Considérons l'utilisation de `p` : à partir de `p`, il faut d'abord utiliser l'opérateur indirection pour obtenir la fonction, soit `*p`. Ensuite, on va appeler la fonction délivrée par l'expression `*p`, pour cela il faut écrire `(*p)()` [2]. Finalement cette expression nous délivre un `int`. La déclaration de `p` s'écrira donc :

```
int (*p)();
```

De la même manière, pour interpréter un type compliqué, il vaut mieux partir d'une utilisation. Soit à interpréter :

```
char (*f())[];
```

Imaginons une utilisation : `(*f(i))[j]`. L'opérateur appel de fonction étant plus prioritaire que l'opérateur indirection, on applique d'abord l'appel de fonction à f. Donc f est une fonction. Ensuite on applique l'opérateur indirection, donc f est une fonction retournant un pointeur. On indexe le résultat, donc f est une fonction retournant un pointeur vers un tableau. Finalement, on voit que f est une fonction retournant un pointeur vers un tableau de `char`.

9.16 Un outil : `cdecl`

Il existe dans le domaine public un petit outil pour manipuler les déclarations du langage C : `cdecl`. On peut le trouver facilement sur Internet en utilisant `xarchie`. Ce programme a été écrit initialement pour HPUX, on le trouve donc généralement classé dans les archives concernant ce système,[3] mais il est facilement portable sur une autre plateforme. Le programme `cdecl` peut traduire une déclaration C en pseudo-anglais (commande `explain`) et vice-versa (commande `declare`).
Exemple d'interaction :

```
osiris(1) cdecl                                         #   appel de cdecl
Type 'help' or '?' for help

cdecl> explain char * argv[]                            #   la question
declare argv as array of pointer to char                #   la réponse

cdecl> declare p as pointer to function returning int   #   la question
int (*p)()                                              #   la réponse

cdecl>
```

2. A l'utilisation il faudra mettre les paramètres effectifs
3. En juillet 96, on peut le trouver sur `hpftp.cict.fr:/hpux/Misc/cdecl-1.0`

Chapitre 10

La bibliothèque standard

Ce chapitre est un aide-mémoire ; il donne la liste exhaustive de toutes les fonctions de la bibliothèque standard, sans donner la sémantique précise de ces fonctions. Pour obtenir plus d'information, utiliser la commande `man` sur une machine UNIX.

10.1 Diagnostic

La fonction `assert` permet de mettre des assertions dans le source du programme. À l'exécution, si le paramètre de `assert` s'évalue à *faux*, le programme est stoppé sur terminaison anormale.

10.2 Manipulation de caractères `<ctype.h>`

Toutes les fonctions ci-dessous permettent de tester une propriété du caractère passé en paramètre.

fonction	le paramètre est
`isalnum`	une lettre ou un chiffre
`isalpha`	une lettre
`iscntrl`	un caractère de commande
`isdigit`	un chiffre décimal
`isgraph`	un caractère imprimable ou le blanc
`islower`	une lettre minuscule
`isprint`	un caractère imprimable (pas le blanc)
`ispunct`	un caractère imprimable pas `isalnum`
`isspace`	un caractère d'espace blanc
`isupper`	une lettre majuscule
`isxdigit`	un chiffre hexadécimal

On dispose également de deux fonctions de conversions entre majuscules et minuscules :

- `tolower` : conversion en minuscule ;
- `toupper` : conversion en majuscule.

10.3 Environnement local `<locale.h>`

Il y a deux fonctions permettant de gérer les conventions nationales concernant l'écriture du point décimal dans les nombres, le signe représentant l'unité monétaire etc. Ces fonctions sont `setlocale` et `localeconv`.

10.4 Mathématiques `<math.h>`

10.4.1 Fonctions trigonométriques et hyperboliques

fonction	sémantique
`acos`	arc cosinus
`asin`	arc sinus
`atan`	arc tangente
`atan2`	arc tangente
`cos`	cosinus
`cosh`	cosinus hyperbolique
`sin`	sinus hyperbolique
`sinh`	sinus
`tan`	tangente
`tanh`	tangente hyperbolique

10.4.2 Fonctions exponentielles et logarithmiques

fonction	sémantique
`exp`	exponentielle
`frexp`	étant donné x, trouve n et p tels que $x = n * 2^p$
`ldexp`	multiplie un nombre par une puissance entière de 2
`log`	logarithme
`log10`	logarithme décimal
`modf`	calcule partie entière et décimale d'un nombre

10.4.3 Fonctions diverses

fonction	sémantique
`ceil`	entier le plus proche par les valeurs supérieures
`fabs`	valeur absolue
`floor`	entier le plus proche par les valeurs inférieures
`fmod`	reste de division
`pow`	puissance
`sqrt`	racine carrée

10.5 Branchements non locaux `<setjmp.h>`

L'instruction `goto` qui ne avons vu au paragraphe 3.15.2 ne permet de réaliser des branchements qu'au sein d'une même procédure. Pour réaliser des branchements à l'extérieur d'une procédure, il faut utiliser `setjmp` et `longjmp`.

10.6 Manipulation des signaux <signal.h>

Deux fonctions permettent d'interagir avec le mécanisme des signaux :

- `signal` permet d'installer une fonction qui sera exécutée sur réception d'un signal ;
- `raise` déclenche un signal chez le processus exécutant.

10.7 Nombre variable de paramètres <stdarg.h>

Si on désire programmer une fonction avec un nombre variable de paramètres, on dispose de trois macros : `va_start`, `va_arg` et `va_end`.

10.8 Entrées sorties <stdio.h>

10.8.1 Opérations sur les fichiers

fonction	description
`remove`	destruction de fichier
`rename`	modification de nom de fichier
`tmpfile`	création d'un fichier temporaire
`tmpnam`	génération de nom approprié pour un fichier temporaire

10.8.2 Accès aux fichiers

fonction	description
`fclose`	fermeture de fichier
`fflush`	écriture sur fichier des buffers en mémoire
`fopen`	ouverture de fichier
`freopen`	ouverture de fichier

10.8.3 Entrées-sorties formattées

fonction	description
`fprintf`	écriture formattée sur flot de données
`fscanf`	lecture formattée sur flot de données
`printf`	écriture formattée sur sortie standard
`scanf`	lecture formattée sur entrée standard
`sprintf`	écriture formattée dans une chaîne de caractères
`sscanf`	lecture formattée depuis une chaîne de caractères
`vfprintf`	variante de `fprintf`
`vprintf`	variante de `printf`
`vsprintf`	variante de `sprintf`

10.8.4 Entrées-sorties caractères

fonction	description
fgetc	lecture d'un caractère
fgets	lecture d'une chaîne de caractères
fputc	écriture d'un caractère
fputs	écriture d'une chaîne de caractères
getc	fgetc implémenté par une macro
getchar	getc sur l'entrée standard
gets	lecture d'une chaîne de caractères sur l'entrée standard
putc	fputc implémenté par une macro
putchar	putc sur la sortie standard
puts	écriture d'une chaîne de caractères sur la sortie standard
ungetc	refoule un caractère (sera lu par la prochain lecture)

10.8.5 Entrées-sorties binaires

Pour lire et écrire des données binaires, on dispose de deux fonctions : `fread` et `fwrite`.

10.8.6 Position dans un fichier

fonction	description
fgetpos	donne la position courante dans un fichier
fseek	permet de se positionner dans un fichier
fsetpos	permet de se positionner dans un fichier
ftell	donne la position courante dans un fichier
rewind	permet de se positionner au début d'un fichier

10.8.7 Gestion des erreurs

fonction	description
clearerr	remet à faux les indicateurs d'erreur et de fin de fichier
feof	test de l'indicateur de fin de fichier
ferror	test de l'indicateur d'erreur
perror	imprime un message d'erreur correspondant à errno

10.9 Utilitaires divers <stdlib.h>

10.9.1 Conversion de nombres

Les fonctions suivantes permettent de convertir des nombres entre la forme chaîne de caractères et la forme binaire.

fonction	description
atof	conversion de chaîne vers double
atoi	conversion de chaîne vers int
atol	conversion de chaîne vers long int
strtod	conversion de chaîne vers double
strtol	conversion de chaîne vers long int
strtoul	conversion de chaîne vers unsigned long int

10.9.2 Génération de nombres pseudo-aléatoires

On dispose de `rand` et `srand`.

10.9.3 gestion de la mémoire

Trois de ces fonctions ont été vues au paragraphe 6.11. La liste exhaustive est `calloc`, `free`, `malloc` et `realloc`.

10.9.4 Communication avec l'environnement

fonction	description
`abort`	terminaison anormale du programme
`atexit`	installe une fonction qui sera exécutée sur terminaison normale du programme
`getenv`	obtention d'une variable d'environnement
`system`	exécution d'un programme

10.9.5 Recherche et tri

Deux fonctions : `bsearch` et `qsort`.

10.9.6 Arithmétique sur les entiers

fonction	description
`abs`	valeur absolue
`div`	obtention de quotient et reste
`labs`	idem `abs` sur des `long int`

10.9.7 Gestion des caractères multi-octets

Les caractères multi-octets permettent de prendre en compte les langues qui ne peuvent se satisfaire de caractères codés sur 8 bits. La liste exhaustive des fonctions est : `mblen`, `mbtowc`, `wctomb`, `mbstowcs`, `wcstombs`.

10.10 Manipulation de chaînes `<string.h>`

On dispose de fonctions pour :

- copier : `memcpy`, `memmove`, `strcpy`, `strncpy` ;
- concaténer : `strcat`, `strncat` ;
- comparer : `memcmp`, `strcmp`, `strcoll`, `strncmp` ;
- transformer : `strxfrm` ;
- rechercher : `memchr`, `strchr`, `strcspn`, `strpbrk`, `strrchr`, `strspn`, `strstr`, `strtok` ;
- initialiser : `memset` ;
- calculer une longueur : `strlen` ;

– obtenir un message d'erreur à partir du numéro de l'erreur : `strerror`.

10.11 Manipulation de la date et de l'heure `<time.h>`

Les fonctions de manipulation de la date et de l'heure sont : `clock`, `difftime`, `mktime`, `time`, `asctime`, `ctime`, `gmtime`, `localtime`, `strftime`.

Annexe A

Les jeux de caractères

Un jeu de caractère est un ensemble de signes typographiques et un codage : à chaque signe est associé un nombre entier qui est son code.

A.1 Les normes

L'histoire des jeux de caractères pour les besoins de l'informatique débute avec le codage EBCDIC (sur 8 bits) utilisé par IBM et le codage ASCII (sur 7 bits) utilisé par le reste du monde. Le code EBCDIC n'est pas une norme, mais le code ASCII est une norme de l'ANSI.

En 1963, l'ISO crée une norme internationale à partir de la norme ASCII et, internationalisation oblige, elle réserve 10 caractères pour des variantes nationales. La France (dans une norme NF) en utilisera 5 pour y caser des lettres accentuées (à ç é è ù) laissant de côté l'accent circonflexe si cher aux académiciens français, le tréma et les ligatures. Cette norme est donc inutilisable pour écrire réellement en français, mais il faut dire à la décharge de l'ISO, qu'en utilisant un code à 7 bits, cela était impossible.

En 1988, l'ISO édite une nouvelle norme de jeu de caractères à 8 bits cette fois pour prendre en compte les particularités des langues européennes. Mais la diversité est telle qu'il n'est pas possible de mettre tous les signes typographiques dans un seul jeu de 256 caractères. Les langues ont donc été regroupées en familles, et la norme ISO-8859 comprend plusieurs jeux de caractères. Celui qui nous intéresse est celui qui est prévu pour les langues d'europe occidentale et qui est référencé ISO-8859-1 ou ISO-LATIN-1. Malheureusement, les français ne sont pas de très bons lobbyistes dans les instances internationales, et aussi incroyable que cela puisse paraître, la ligature œ du français a été oubliée !

Pour en savoir plus

Il existe un excellent article de Jacques André et Michel Goosens sur les problèmes de normalisation de codage de caractères, librement accessible via l'URL :
`http://www.univ-rennes1.fr/pub/gut/publications` . Il s'agit de la revue « les cahiers de GUTenberg » , et l'article en question est dans le cahier 20. Pour le problème de la ligature œ, voir le cahier 25.

A.2 Le code ascii

Le jeu de caractères est formé d'un ensemble de caractères de commandes et de caractères graphiques. L'ensemble des caractères de commande est formé de six familles :

- commandes de format

commande	nom
carriage return	CR
line feed	LF
backspace	BS
horizontal tabulation	HT
vertical tabulation	VT
space	SP
form feed	FF

Le nom *carriage return* arrive tout droit de l'époque des machines à écrire, où la position d'écriture était fixe et où le papier était porté sur un chariot (*carriage*) mobile. Le caractère *carriage return* est la commande permettant de mettre la position d'écriture en début de ligne, sans changer de ligne. Le caractère *line feed* met la position d'écriture sur la ligne suivante, sans aller en début de ligne. Pour obtenir l'effet de « passage à la ligne » , il faut donc un caractère *carriage return* suivi d'un caractère *line feed* (ou l'inverse). Dans le système UNIX, le caractère choisi par convention comme signifiant « passage à la ligne » est le caractère *line feed*, et c'est à la charge des pilotes de périphériques de remplacer ce caractère logique par la suite de caractères nécessaires pour obtenir un passage à la ligne suivante. Prenons le cas d'un pilote de terminal écran clavier :

– en entrée : la convention habituelle est de faire un passage à la ligne an appuyant sur la touche *carriage return*. Le pilote de terminal :
  1. envoie au programme qui réalise la lecture un *line feed*.
  2. envoie à l'écran (en tant qu'écho de ce *carriage return*) la séquence *line feed* suivi de *carriage return*.

– en sortie : le pilote de terminal transforme les *line feed* en *line feed* suivi de *carriage return*

Par abus de langage, dans le monde C et/ou UNIX, on utilise souvent le terme de *newline* pour désigner en réalité *line feed*. Mais qu'il soit bien clair que la norme ANSI ne comporte pas de caractère appelé *newline*.

- commandes d'extension du code

commande	nom
shift out	SO
shift in	SI
escape	ESC

Le caractère *escape* a été largement utilisé par les concepteurs de terminaux écran-clavier et d'imprimantes pour augmenter le nombre de commandes. La technique consiste à définir des *séquences d'échappement* formées du caractère *escape* suivi d'un certains nombre de caractères ordinaires qui perdent leur signification habituelle. Voici quelques *séquences d'échappement* du terminal écran-clavier VT100 :

séquence	sémantique
escape `[2A`	monter le curseur de 2 lignes
escape `[4B`	descendre le curseur de 4 lignes
escape `[3C`	décaler le curseur de 3 positions vers la droite
escape `[1D`	décaler le curseur de 1 position vers la gauche

- commande de séparation

commande	nom
file separator	FS
group separator	GS
record separator	RS
unit separator	US
end of medium	EM

Ces caractères ont pour but de séparer les différentes unités d'information sur bandes magnétiques. Ils sont obsolètes de nos jours, les programmes d'archivage (`tar`, `cpio`) utilisant leur propre format sans faire appel à ces caractères.

- commandes pour la communication synchrone

commande	nom
start of header	SOH
start of text	STX
end of text	ETX
end of transmission	EOT
end of transmitted block	ETB
enquiry	ENQ
positive acknowledge	ACK
negative acknowledge	NAK
synchronisation	SYN
data link escape	DLE
null	NUL

Les 10 premières commandes ont été crées pour construire des trames de communication entre machines reliées par des lignes synchrones. Elles sont complètement obsolètes de nos jours, où les communications se font grâce à des réseaux dont les trames n'utilisent pas ces caractères.

La dernière commande *null* était utile à l'époque des téléimprimeurs dont le temps de retour du chariot était plus grand que le temps d'impression d'un caractère quelconque. Après avoir envoyé un *carriage return*, il fallait envoyer plusieurs *null* (en fonction de la vitesse de la ligne) pour être sûr que le chariot était bien revenu en début de ligne !

- commandes de périphérique

commande	nom
device control 1	DC1
device control 2	DC2
device control 3	DC3
device control 4	DC4

Ces caractères ont été prévus pour donner des ordres spécifiques à certains périphériques. A l'époque des téléimprimeurs, ceux-ci possédaient un lecteur-perforateur de ruban papier. Les codes *device control* étaient utilisés pour commander ce lecteur-perforateur.

De nos jours *device control 3* et *device control 1* sont utilisés sous les noms respectifs de XON et XOFF pour réaliser du contrôle de flux. Les caractères *device control 3* et *device control 1* sont affectés aux touches *Control-q* et *Control-s* du clavier. Lorsqu'un pilote de terminal écran-clavier gère le contrôle de flux, l'utilisateur peut taper *Control-s* pour faire stopper une sortie trop rapide (pour se donner le temps de la lire sur l'écran), et la faire continuer en tapant *Control-q*.

- commandes diverses

commande	nom
cancel	CAN
substitute	SUB
delete	DEL
bell	BEL

Il y a deux caractères qui sont utilisés couramment pour réaliser la fonction d'effacement du caractère (erroné) précédent : *back space* et *delete*. En fonction du caractère qui est le plus facile à taper sur son clavier, l'utilisateur désirera choisir l'un ou l'autre. Le caractère *back space* peut sur tout clavier s'obtenir par *Control-h*, alors qu'il n'y a pas de *Control-quelque-chose* correspondant au caractère *delete*. Selon les claviers, il peut y avoir une touche marquée *back space*, et/ou une touche marquée *delete*, ou une touche marquée ← qui génère *back space* ou *delete*, et qui peut, ou ne peut pas, être configurée par le *set-up* du terminal pour générer au choix *back space* ou *delete* !

Un utilisateur UNIX utilise la commande `stty` pour indiquer au système d'exploitation le caractère qu'il désire pour réaliser la fonction d'effacement de caractère.

A.2.1 Les codes ascii en octal

code	0	1	2	3	4	5	6	7
000	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL
010	BS	HT	LF	VT	NP	CR	SO	SI
020	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB
030	CAN	EM	SUB	ESC	FS	GS	RS	US
040	SP	!	"	#	$	%	&	’
050	(	)	*	+	,	-	.	/
060	0	1	2	3	4	5	6	7
070	8	9	:	;	<	=	>	?
100	@	A	B	C	D	E	F	G
110	H	I	J	K	L	M	N	O
120	P	Q	R	S	T	U	V	W
130	X	Y	Z	[	\	]	^	_
140	‘	a	b	c	d	e	f	g
150	h	i	j	k	l	m	n	o
160	p	q	r	s	t	u	v	w
170	x	y	z	{	|	}	~	DEL

A.2.2 Les codes ascii en hexadécimal

code	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x00	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	NP	CR	SO	SI
0x10	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
0x20	SP	!	"	#	$	%	&	’	(	)	*	+	,	-	.	/
0x30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
0x40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0x50	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
0x60	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
0x70	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

A.2.3 Les codes ascii en décimal

code	0	1	2	3	4	5	6	7	8	9
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT
10	LF	VT	NP	CR	SO	SI	DLE	DC1	DC2	DC3
20	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS
30	RS	US	SP	!	"	#	$	%	&	'
40	(	)	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[	\	]	^	_	`	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{	|	}	~	DEL		

A.3 Les codes ISO-Latin-1

octal	0	1	2	3	4	5	6	7
0000	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL
0010	BS	HT	LF	VT	NP	CR	SO	SI
0020	DLE	DC1	DC2	DC3	DC4	NAQ	SYN	ETB
0030	CAN	EM	SUB	ESC	FS	GS	RS	US
0040	SP	!	"	#	$	%	&	’
0050	(	)	*	+	,	−	.	/
0060	0	1	2	3	4	5	6	7
0070	8	9	:	;	<	=	>	?
0100	@	A	B	C	D	E	F	G
0110	H	I	J	K	L	M	N	O
0120	P	Q	R	S	T	U	V	W
0130	X	Y	Z	[	\	]	^	_
0140	‘	a	b	c	d	e	f	g
0150	h	i	j	k	l	m	n	o
0160	p	q	r	s	t	u	v	w
0170	x	y	z	{	|	}	~	DEL
0200								
0210								
0220	ı	`	´	ˆ	˜	¯	˘	˙
0230	¨		˚	¸		˝	˛	ˇ
0240		¡	¢	£	¤	¥	¦	§
0250	¨	©	ª	«	¬	-	®	¯
0260	°	±	²	³	´	µ	¶	·
0270	¸	¹	º	»	¼	½	¾	¿
0300	À	Á	Â	Ã	Ä	Å	Æ	Ç
0310	È	É	Ê	Ë	Ì	Í	Î	Ï
0320	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×
0330	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
0340	à	á	â	ã	ä	å	æ	ç
0350	è	é	ê	ë	ì	í	î	ï
0360	ð	ñ	ò	ó	ô	õ	ö	÷
0370	ø	ù	ú	û	ü	ý	þ	ÿ

Annexe B

Bibliographie

Il existe de très nombeux livres sur le langage C. Il suffit de se rendre dans une FNAC quelconque pour en trouver des rayons entiers. Je ne donnerai ici que les livres qui sont extraordinaires pour une raison ou une autre.

[1] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language.* Prentice Hall, seconde édition, 1988.

Le livre sur C écrit par les concepteurs du langage. La seconde édition est conforme au standard ANSI, alors que la première édition (même titre, mêmes auteurs) définissait le langage C dans sa version dite “Kernighan et Ritchie”.

[2] Samuel P. Harbison et Guy L. Steele. *C a reference manual.* Prentice hall, quatrième édition, 1995.

Il y a un consensus sur Internet pour estimer que ce livre est la meilleure référence sur le langage C. C'est le livre dont un programmeur doit disposer dès qu'il a dépassé le stade d'apprentissage du langage.

[3] Herbert Schildt. *The annotated ANSI C standard.* Osborne McGraw-Hill, 1993

L'idée de H. Schildt était excellente : donner le texte complet de la norme accompagné de commentaires permettant de l'éclairer. Malheureusement, le résultat est considéré comme étant très mauvais par pratiquement tout le monde sur Internet : les annotations de l'auteur sont considérées comme étant sans intéret. Il y a cependant une raison de se procurer ce livre : obtenir au prix d'un livre le texte officiel d'une norme qui coute très cher si on se la procure directement auprès des organismes normalisateurs.

[4] International Standard ISO/IEC 9899:1990 *Programming languages – C*

Pour les personnes qui doivent disposer du texte officiel de la norme avec les derniers amendements. On peut se procurer ce texte auprès de l'AFNOR. Consulter `http://www.afnor.fr`.

[5] Peter Van der Linden *Expert C programming.* Sun Soft Press - A Prentice Hall Title

Ce livre n'est ni un livre pour apprendre le langage, ni un manuel de référence pour programmeur confirmé. C'est un mélange d'anecdotes, d'études fouillées de certains points difficiles du langage, de défis de programmation, de récit de bugs désastreux etc. Se lit davantage comme un roman que comme un livre technique.

Annexe C

Ressources Internet

Les ressources Internet se composent essentiellement de :

- deux forums de discussion :
  - `comp.lang.c` : forum international (donc en langue anglaise) de discussion sur le langage C.
  - `fr.comp.lang.c` : forum français de discussion sur le langage C.
- deux FAQ (Frequently Asked Questions) :
  - La faq de `comp.lang.c` maintenue par Steve Summit dont l'URL est `ftp://rtfm.mit.edu:/pub/usenet/news.answers/C-faq/faq`. C'est le recueil des questions les plus fréquemment posées dans le forum `comp.lang.c`.
  - Un document intitulé *Learn C/C++ today (a list of resources/tutorials)* dont l'URL est `ftp://rtfm.mit.edu:/pub/usenet/news.answers/C-faq/learn-c-cpp-today`. On y trouve les URL de ressources librement accessibles sur Internet, à savoir : des documents à imprimer, des documents à consulter avec un browser WEB, des exemples de programmes, une bibliographie commentée.

Annexe D

La grammaire

D.1 Les unités lexicales

R_1 *unité-lexicale* :
- ⇒ *mot-clé*
- ⇒ *identificateur*
- ⇒ *constante*
- ⇒ *chaîne-littérale*
- ⇒ *opérateur*
- ⇒ *ponctuation*

R_2 *unité-lexicale-du-pp* :
- ⇒ *nom-fichier-inclusion*
- ⇒ *identificateur*
- ⇒ *nombre-du-pp*
- ⇒ *constante-caractère*
- ⇒ *chaîne-littérale*
- ⇒ *opérateur*
- ⇒ *ponctuation*
- ⇒ tout caractère non blanc qui ne peut être une des entités précédentes

D.2 Les mots-clés

R_3 *mot-clé* : un parmi

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

D.3 Les identificateurs

R_4 *identificateur* :

⇒ *non-chiffre*

⇒ *identificateur non-chiffre*

⇒ *identificateur chiffre*

R_5 *non-chiffre* : un parmi

```
_ a b c d e f g h i j k l m
  n o p q r s t u v w x y z
  A B C D E F G H I J K L M
  N O P Q R S T U V W X Y Z
```

R_6 *chiffre* : un parmi

```
0 1 2 3 4 5 6 7 8 9
```

D.4 Les constantes

R_7 *constante* :

⇒ *constante-flottante*

⇒ *constante-entière*

⇒ *constante-d-énumération*

⇒ *constante-caractère*

R_8 *constante-flottante* :

⇒ *partie-fractionnaire* *partie-exposant*$_{option}$ *suffixe-flottant*$_{option}$

⇒ *séquence-de-chiffres* *partie-exposant* *suffixe-flottant*$_{option}$

R_9 *partie-fractionnaire* :

⇒ *séquence-de-chiffres*$_{option}$ `.` *séquence-de-chiffres*

⇒ *séquence-de-chiffres* `.`

R_{10} *partie-exposant* :

⇒ `e` *signe*$_{option}$ *séquence-de-chiffres*

⇒ `E` *signe*$_{option}$ *séquence-de-chiffres*

R_{11} *signe* : un parmi

```
+ -
```

R_{12} *séquence-de-chiffres* :

⇒ *chiffre*

⇒ *séquence-de-chiffres chiffre*

R_{13} *suffixe-flottant* : un parmi

```
f l F L
```

R_{14} *constante-entière* :

⇒ *constante-décimale* *suffixe-entier*$_{option}$

⇒ *constante-octale* *suffixe-entier*$_{option}$

⇒ *constante-hexadécimale* *suffixe-entier*$_{option}$

R_{15} *constante-décimale* :

⇒ *chiffre-non-nul*

⇒ *constante-décimale* *chiffre*

R_{16} *constante-octale* :

⇒ `0`

⇒ *constante-octale* *chiffre-octal*

R_{17} *constante-hexadécimale* :

⇒ `0x` *chiffre-hexadécimal*

⇒ `0X` *chiffre-hexadécimal*

⇒ *constante-hexadécimale* *chiffre-hexadécimal*

R_{18} *chiffre-non-nul* : un parmi

`1 2 3 4 5 6 7 8 9`

R_{19} *chiffre-octal* : un parmi

`0 1 2 3 4 5 6 7`

R_{20} *chiffre-hexadécimal* : un parmi

`0 1 2 3 4 5 6 7 8 9`

`a b c d e f A B C D E F`

R_{21} *suffixe-entier* :

⇒ *suffixe-non-signé* *suffixe-long*$_{option}$

⇒ *suffixe-long* *suffixe-non-signé*$_{option}$

R_{22} *suffixe-non-signé* : un parmi

`u U`

R_{23} *suffixe-long* : un parmi

`l L`

R_{24} *constante-d-énumération* :

⇒ *identificateur*

R_{25} *constante-caractère* :

⇒ `'` *séquence-de-caractères-c* `'`

⇒ `L'` *séquence-de-caractères-c* `'`

R_{26} *séquence-de-caractères-c* :

⇒ *caractère-c*

⇒ *séquence-de-caractères-c* *caractère-c*

R$_{27}$ *caractère-c* :
⇒ n'importe quel membre du jeu de caractères source sauf le quote (`'`) le *backslash* (`\`) ou le *newline*
⇒ *séquence-d-échappement*

R$_{28}$ *séquence-d-échappement* :
⇒ *séquence-d-échappement-simple*
⇒ *séquence-d-échappement-octale*
⇒ *séquence-d-échappement-hexadécimale*

R$_{29}$ *séquence-d-échappement-simple* : un parmi
`\'` `\"` `\?` `\\`
`\a` `\b` `\f` `\n` `\r` `\t` `\v`

R$_{30}$ *séquence-d-échappement-octale* :
⇒ `\` *chiffre-octal*
⇒ `\` *chiffre-octal* *chiffre-octal*
⇒ `\` *chiffre-octal* *chiffre-octal* *chiffre-octal*

R$_{31}$ *séquence-d-échappement-hexadécimale* :
⇒ `\x` *chiffre-hexadécimal*
⇒ *séquence-d-échappement-hexadécimale* *chiffre-hexadécimal*

D.5 Les chaînes littérales

R$_{32}$ *chaîne-littérale* :
⇒ `"` *séquence-de-caractères-s*$_{option}$ `"`
⇒ `L"` *séquence-de-caractères-s*$_{option}$ `"`

R$_{33}$ *séquence-de-caractères-s* :
⇒ *caractère-s*
⇒ *séquence-de-caractères-s* *caractère-s*

R$_{34}$ *caractère-s* :
⇒ n'importe quel membre du jeu de caractères source sauf le *double quote* (`"`) le *backslash* (`\`) ou le *newline*
⇒ *séquence-d-échappement*

D.6 Les opérateurs

R$_{35}$ *opérateur* : un parmi
`[` `]` `(` `)` `.` `->`

```
++  --  &  *  +  -  ~  !  sizeof
/  %  <<  >>  <  >  <=  >=  ==  !=  ^  |  &&  ||
?  :
=  *=  /=  %=  +=  -=  <<=  >>=  &=  ^=  |=
,  #  ##
```

D.7 La ponctuation

R$_{36}$ *ponctuation* : un parmi

```
[  ]  (  )  {  }  *  ,  :  =  ;  ...  #
```

D.8 Nom de fichier d'inclusion

R$_{37}$ *nom-fichier-inclusion* :
- ⇒ `<` *séquence-de-caractères-h* `>`
- ⇒ `"` *séquence-de-caractères-q* `"`

R$_{38}$ *séquence-de-caractères-h* :
- ⇒ *caractère-h*
- ⇒ *séquence-de-caractères-h* *caractère-h*

R$_{39}$ *caractère-h* :
- ⇒ n'importe quel membre du jeu de caractères source sauf `>` ou le *newline*

R$_{40}$ *séquence-de-caractères-q* :
- ⇒ *caractère-q*
- ⇒ *séquence-de-caractères-q* *caractère-q*

R$_{41}$ *caractère-q* :
- ⇒ n'importe quel membre du jeu de caractères source sauf `"` ou le *newline*

D.9 Les nombres du préprocesseur

R$_{42}$ *nombre-du-pp* :
- ⇒ *chiffre*
- ⇒ `.` *chiffre*
- ⇒ *nombre-du-pp* *chiffre*
- ⇒ *nombre-du-pp* *non-chiffre*
- ⇒ *nombre-du-pp* `e` *signe*
- ⇒ *nombre-du-pp* `E` *signe*
- ⇒ *nombre-du-pp* `.`

D.10 Les expressions

R43 *expression-primaire* :
- ⇒ *identificateur*
- ⇒ *constante*
- ⇒ *chaîne-littérale*
- ⇒ `(` *expression* `)`

R44 *expression-postfixée* :
- ⇒ *expression-primaire*
- ⇒ *expression-postfixée* `[` *expression* `]`
- ⇒ *expression-postfixée* `(` *liste-d-expressions-paramètres*$_{option}$ `)`
- ⇒ *expression-postfixée* `.` *identificateur*
- ⇒ *expression-postfixée* `->` *identificateur*
- ⇒ *expression-postfixée* `++`
- ⇒ *expression-postfixée* `--`

R45 *liste-d-expressions-paramètres* :
- ⇒ *expression-affectation*
- ⇒ *liste-d-expressions-paramètres* `,` *expression-affectation*

R46 *expression-unaire* :
- ⇒ *expression-postfixée*
- ⇒ `++` *expression-unaire*
- ⇒ `--` *expression-unaire*
- ⇒ *opérateur-unaire* *expression-cast*
- ⇒ `sizeof` *expression-unaire*
- ⇒ `sizeof (` *nom-de-type* `)`

R47 *opérateur-unaire* : un parmi

`&  *  +  -  ~  !`

R48 *expression-cast* :
- ⇒ *expression-unaire*
- ⇒ `(` *nom-de-type* `)` *expression-cast*

R49 *expression-multiplicative* :
- ⇒ *expression-cast*
- ⇒ *expression-multiplicative* `*` *expression-cast*
- ⇒ *expression-multiplicative* `/` *expression-cast*
- ⇒ *expression-multiplicative* `%` *expression-cast*

R50 *expression-additive* :
- ⇒ *expression-multiplicative*
- ⇒ *expression-additive* `+` *expression-multiplicative*
- ⇒ *expression-additive* `-` *expression-multiplicative*

R$_{51}$ *expression-décalage* :

⇒ *expression-additive*
⇒ *expression-décalage* `<<` *expression-additive*
⇒ *expression-décalage* `>>` *expression-additive*

R$_{52}$ *expression-relation* :

⇒ *expression-décalage*
⇒ *expression-relation* `<` *expression-décalage*
⇒ *expression-relation* `>` *expression-décalage*
⇒ *expression-relation* `<=` *expression-décalage*
⇒ *expression-relation* `>=` *expression-décalage*

R$_{53}$ *expression-égalité* :

⇒ *expression-relation*
⇒ *expression-égalité* `==` *expression-relation*
⇒ *expression-égalité* `!=` *expression-relation*

R$_{54}$ *expression*-ET :

⇒ *expression-égalité*
⇒ *expression*-ET `&` *expression-égalité*

R$_{55}$ *expression*-OU-*exclusif* :

⇒ *expression*-ET
⇒ *expression*-OU-*exclusif* `^` *expression*-ET

R$_{56}$ *expression*-OU-*inclusif* :

⇒ *expression*-OU-*exclusif*
⇒ *expression*-OU-*inclusif* `|` *expression*-OU-*exclusif*

R$_{57}$ *expression*-ET-*logique* :

⇒ *expression*-OU-*inclusif*
⇒ *expression*-ET-*logique* `&&` *expression*-OU-*inclusif*

R$_{58}$ *expression*-OU-*logique* :

⇒ *expression*-ET-*logique*
⇒ *expression*-OU-*logique* `||` *expression*-ET-*logique*

R$_{59}$ *expression-conditionnelle* :

⇒ *expression*-OU-*logique*
⇒ *expression*-OU-*logique* `?` *expression* `:` *expression-conditionnelle*

R$_{60}$ *expression-affectation* :

⇒ *expression-conditionnelle*
⇒ *expression-unaire* *opérateur-affectation* *expression-affectation*

R$_{61}$ *opérateur-affectation* : un parmi

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

R$_{62}$ *expression* :

⇒ *expression-affectation*

⇒ *expression* , *expression-affectation*

R$_{63}$ *expression-constante* :

⇒ *expression-conditionnelle*

D.11 Les déclarations

R$_{64}$ *déclaration* :

⇒ *spécificateurs-de-déclaration* *liste-de-déclarateurs-init*$_{option}$;

R$_{65}$ *spécificateurs-de-déclaration* :

⇒ *spécificateur-de-classe-mémoire* *spécificateurs-de-déclaration*$_{option}$

⇒ *spécificateur-de-type* *spécificateurs-de-déclaration*$_{option}$

⇒ *qualificatif-de-type* *spécificateurs-de-déclaration*$_{option}$

R$_{66}$ *liste-de-déclarateurs-init* :

⇒ *déclarateur-init*

⇒ *liste-de-déclarateurs-init* , *déclarateur-init*

R$_{67}$ *déclarateur-init* :

⇒ *déclarateur*

⇒ *déclarateur* `=` *initialisateur*

R$_{68}$ *spécificateur-de-classe-mémoire* :

⇒ `auto`

⇒ `extern`

⇒ `static`

⇒ `register`

⇒ `typedef`

R$_{69}$ *spécificateur-de-type* :

⇒ `void`

⇒ `char`

⇒ `short`

⇒ `int`

⇒ `long`

⇒ `float`

⇒ `double`

⇒ `signed`

⇒ `unsigned`

⇒ *spécificateur-de-struct-ou-union*

⇒ *spécificateur-d-énumération*

⇒ *nom-de-typedef*

R_{70} *spécificateur-de-struct-ou-union* :
⇒ *struct-ou-union* *identificateur*$_{option}$ `{` *liste-de-déclarations-de-struct* `}`
⇒ *struct-ou-union* *identificateur*

R_{71} *struct-ou-union* :
⇒ `struct`
⇒ `union`

R_{72} *liste-de-déclarations-de-struct* :
⇒ *déclaration-de-struct*
⇒ *liste-de-déclarations-de-struct* *déclaration-de-struct*

R_{73} *déclaration-de-struct* :
⇒ *liste-de-spécificateurs-et-qualificatifs* *liste-de-déclarateurs-de-struct* `;`

R_{74} *liste-de-spécificateurs-et-qualificatifs* :
⇒ *spécificateur-de-type* *liste-de-spécificateurs-et-qualificatifs*$_{option}$
⇒ *qualificatif-de-type* *liste-de-spécificateurs-et-qualificatifs*$_{option}$

R_{75} *liste-de-déclarateurs-de-struct* :
⇒ *déclarateur-de-struct*
⇒ *liste-de-déclarateurs-de-struct* `,` *déclarateur-de-struct*

R_{76} *déclarateur-de-struct* :
⇒ *déclarateur*
⇒ *déclarateur*$_{option}$ `:` *expression-constante*

R_{77} *spécificateur-d-énumération* :
⇒ `enum` *identificateur*$_{option}$ `{` *liste-d-énumérateurs* `}`
⇒ `enum` *identificateur*

R_{78} *liste-d-énumérateurs* :
⇒ *énumérateur*
⇒ *liste-d-énumérateurs* `,` *énumérateur*

R_{79} *énumérateur* :
⇒ *constante-d-énumération*
⇒ *constante-d-énumération* `=` *expression-constante*

R_{80} *qualificatif-de-type* :
⇒ `const`
⇒ `volatile`

R_{81} *déclarateur* :
⇒ *pointeur*$_{option}$ *déclarateur-direct*

R_{82} *déclarateur-direct* :

⇒ *identificateur*
⇒ (*déclarateur*)
⇒ *déclarateur-direct* [*expression-constante*$_{option}$]
⇒ *déclarateur-direct* (*liste-de-types-de-paramètres*)
⇒ *déclarateur-direct* (*liste-d-identificateurs*$_{option}$)

R_{83} *pointeur* :

⇒ * *liste-de-qualificatifs-de-types*$_{option}$
⇒ * *liste-de-qualificatifs-de-types*$_{option}$ *pointeur*

R_{84} *liste-de-qualificatifs-de-types* :

⇒ *qualificatif-de-type*
⇒ *liste-de-qualificatifs-de-types* *qualificatif-de-type*

R_{85} *liste-de-types-de-paramètres* :

⇒ *liste-de-paramètres*
⇒ *liste-de-paramètres* , ...

R_{86} *liste-de-paramètres* :

⇒ *déclaration-de-paramètre*
⇒ *liste-de-paramètres* , *déclaration-de-paramètre*

R_{87} *déclaration-de-paramètre* :

⇒ *spécificateurs-de-déclaration* *déclarateur*
⇒ *spécificateurs-de-déclaration* *déclarateur-abstrait*$_{option}$

R_{88} *liste-d-identificateurs* :

⇒ *identificateur*
⇒ *liste-d-identificateurs* , *identificateur*

R_{89} *nom-de-type* :

⇒ *liste-de-spécificateurs-et-qualificatifs* *déclarateur-abstrait*$_{option}$

R_{90} *déclarateur-abstrait* :

⇒ *pointeur*
⇒ *pointeur*$_{option}$ *déclarateur-abstrait-direct*

R_{91} *déclarateur-abstrait-direct* :

⇒ (*déclarateur-abstrait*)
⇒ *déclarateur-abstrait-direct*$_{option}$ [*expression-constante*$_{option}$]
⇒ *déclarateur-abstrait-direct*$_{option}$ (*liste-de-types-de-paramètres*$_{option}$)

R_{92} *nom-de-typedef* :

⇒ *identificateur*

R$_{93}$ *initialisateur* :

⇒ *expression-affectation*

⇒ `{` *liste-d-initialisateurs* `}`

⇒ `{` *liste-d-initialisateurs* `,` `}`

R$_{94}$ *liste-d-initialisateurs* :

⇒ *initialisateur*

⇒ *liste-d-initialisateurs* `,` *initialisateur*

D.12 Les instructions

R$_{95}$ *instruction* :

⇒ *instruction-étiquettée*

⇒ *instruction-composée*

⇒ *instruction-expression*

⇒ *instruction-sélection*

⇒ *instruction-itération*

⇒ *instruction-saut*

R$_{96}$ *instruction-étiquettée* :

⇒ *identificateur* `:` *instruction*

⇒ `case` *expression-constante* `:` *instruction*

⇒ `default :` *instruction*

R$_{97}$ *instruction-composée* :

⇒ `{` *liste-de-déclarations*$_{option}$ *liste-d-instructions*$_{option}$ `}`

R$_{98}$ *liste-de-déclarations* :

⇒ *déclaration*

⇒ *liste-de-déclarations* *déclaration*

R$_{99}$ *liste-d-instructions* :

⇒ *instruction*

⇒ *liste-d-instructions* *instruction*

R$_{100}$ *instruction-expression* :

⇒ *expression*$_{option}$ `;`

R$_{101}$ *instruction-sélection* :

⇒ `if (` *expression* `)` *instruction*

⇒ `if (` *expression* `)` *instruction* `else` *instruction*

⇒ `switch (` *expression* `)` *instruction*

R$_{102}$ *instruction-itération* :

⇒ `while (` *expression* `)` *instruction*

⇒ `do` *instruction* `while (` *expression* `) ;`
⇒ `for (` $expression_{option}$ `;` $expression_{option}$ `;` $expression_{option}$ `)` *instruction*

R$_{103}$ *instruction-saut*:
⇒ `goto` *identificateur* `;`
⇒ `continue ;`
⇒ `break ;`
⇒ `return` $expression_{option}$ `;`

D.13 Définitions externes

R$_{104}$ *unité-de-compilation*:
⇒ *déclaration-externe*
⇒ *unité-de-compilation* *déclaration-externe*

R$_{105}$ *déclaration-externe*:
⇒ *définition-de-fonction*
⇒ *déclaration*

R$_{106}$ *définition-de-fonction*:
⇒ *spécificateurs-de-déclaration*$_{option}$ *déclarateur* *liste-de-déclarations*$_{option}$ *instruction-composée*

D.14 Directives du préprocesseur

R$_{107}$ *fichier-du-pp*:
⇒ *groupe*$_{option}$

R$_{108}$ *groupe*:
⇒ *partie-de-groupe*
⇒ *groupe* *partie-de-groupe*

R$_{109}$ *partie-de-groupe*:
⇒ *liste-d-unités-lexicales-du-pp*$_{option}$ *newline*
⇒ *section-if*
⇒ *ligne-directive*

R$_{110}$ *section-if*:
⇒ *groupe-if* *liste-de-groupes-elif*$_{option}$ *groupe-else*$_{option}$ *ligne-endif*

R$_{111}$ *groupe-if*:
⇒ `# if` *expression-constante* *newline* *groupe*$_{option}$
⇒ `# ifdef` *identificateur* *newline* *groupe*$_{option}$

$\Rightarrow$ `#` `ifndef` *identificateur* *newline* *groupe*$_{option}$

R$_{112}$ *liste-de-groupes-elif*:
$\Rightarrow$ *groupe-elif*
$\Rightarrow$ *liste-de-groupes-elif* *groupe-elif*

R$_{113}$ *groupe-elif*:
$\Rightarrow$ `#` `elif` *expression-constante* *newline* *groupe*$_{option}$

R$_{114}$ *groupe-else*:
$\Rightarrow$ `#` `else` *newline* *groupe*$_{option}$

R$_{115}$ *ligne-endif*:
$\Rightarrow$ `#` `endif` *newline*

R$_{116}$ *ligne-directive*:
$\Rightarrow$ `#` `include` *liste-d-unités-lexicales-du-pp* *newline*
$\Rightarrow$ `#` `define` *identificateur* *remplacement* *newline*
$\Rightarrow$ `#` `define` *identificateur* *parenthèse-g* *liste-d-identificateurs*$_{option}$ *remplacement* *newline*
$\Rightarrow$ `#` `undef` *identificateur* *newline*
$\Rightarrow$ `#` `line` *liste-d-unités-lexicales-du-pp* *newline*
$\Rightarrow$ `#` `error` *liste-d-unités-lexicales-du-pp*$_{option}$ *newline*
$\Rightarrow$ `#` `pragma` *liste-d-unités-lexicales-du-pp*$_{option}$ *newline*
$\Rightarrow$ `#` *newline*

R$_{117}$ *parenthèse-g*:
$\Rightarrow$ le caractère `(` non précédé d'un espace blanc

R$_{118}$ *remplacement*:
$\Rightarrow$ *liste-d-unités-lexicales-du-pp*$_{option}$

R$_{119}$ *liste-d-unités-lexicales-du-pp*:
$\Rightarrow$ *unité-lexicale-du-pp*
$\Rightarrow$ *liste-d-unités-lexicales-du-pp* *unité-lexicale-du-pp*

R$_{120}$ *newline*:
$\Rightarrow$ le caractère séparateur de lignes

D.15 Références croisées de la grammaire

caractère-c	**27** 26
caractère-h	**39** 38
caractère-q	**41** 40
caractère-s	**34** 33
chaîne-littérale	**2** 1, 2, 43
chiffre	**6** 4, 42
chiffre-hexadécimal	**20** 17, 31
chiffre-non-nul	**18** 15
chiffre-octal	**19** 16, 30
constante	**7** 1, 43
constante-caractère	**25** 2, 7
constante-d-énumération	**24** 7, 79
constante-décimale	**15** 14, 15
constante-entière	**14** 7
constante-flottante	**8** 7
constante-hexadécimale	**17** 14, 17
constante-octale	**16** 14
déclarateur	**81** 67, 76, 82 87, 106
déclarateur-abstrait	**90** 87, 89, 91
déclarateur-abstrait-direct	**91** 90, 91
déclarateur-de-struct	**76** 75
déclarateur-direct	**82** 81, 82
déclarateur-init	**67** 66
déclaration	**64** 98, 105
déclaration-de-paramètre	**87** 86
déclaration-de-struct	**73** 72
déclaration-externe	**105** 104
définition-de-fonction	**106** 105
énumérateur	**79** 78
expression	**62** 43, 44, 59, 62, 100, 101, 102, 103
expression-additive	**50** 50, 51
expression-affectation	**60** 45, 60, 62, 93
expression-cast	**48** 46, 48, 49
expression-conditionnelle	**59** 59, 60, 63
expression-constante	**63** 76, 79, 82, 91, 96, 111, 113
expression-décalage	**51** 51, 52
expression-égalité	**53** 53, 54
expression-ET	**54** 54, 55
expression-ET-logique	**57** 57, 58
expression-multiplicative	**49** 49, 50
expression-OU-exclusif	**55** 55, 56
expression-OU-inclusif	**56** 56, 57
expression-OU-logique	**58** 58, 59
expression-postfixée	**44** 44, 46
expression-primaire	**43** 44

Annexe E

Un bestiaire de types

E.1 Les types de base

formes équivalentes	forme préférée	sémantique

`short`
`short int`
`signed short`
`signed short int`
	`short int`	entier court

`int`
`signed`
`signed int`
	`int`	entier

`long`
`long int`
`signed long`
`signed long int`
	`long int`	entier long

`unsigned short` `unsigned short int`	`unsigned short int`	entier court non signé
`unsigned` `unsigned int`	`unsigned int`	entier non signé
`unsigned long` `unsigned long int`	`unsigned long int`	entier long non signé
`char`		caractère
`signed char`		caractère signé
`unsigned char`		caractère non signé
`float`		flottant simple précision
`double`		flottant double précision
`long double`		flottant quadruple précision

E.2 Les tableaux

`int tab[10] ;`	tableau de 10 entiers
`int tab[5] = { 10, 11, 12, 13, 14};`	tableau de 5 entiers avec initialisation
`char mes[80];`	tableau de 80 caractères
`char mess[] = "Erreur de syntaxe";`	tableau de caractères avec initialisation
`int t[24][80];`	tableau à deux dimensions

`int t1[2][3] = {`
`   {0, 1, 2},`
`   { 10, 11, 12}`
`};`
	tableau à deux dimensions avec initialisation

`int *t[10];`	tableau de pointeurs vers des entiers
`int i, j, k;` `int *t[3] = {&i, &j, &k};`	tableau de pointeurs vers des entiers avec initialisation

`char *t[] = {`
`   "lundi",`
`   "mardi",`
`   "mercredi",`
`   };`
	tableau de pointeurs vers des caractères avec initialisation

`struct complex { float x,y; };` `struct complex t[10];`	tableau de structures

`struct complex { float x,y; };`
`struct complex t[3] = {`
`   {1.5, 2.4},`
`   {1,   3.2},`
`   {0.7, 3}`
`};`
	tableau de structures avec initialisation

`struct complex { float x,y; };` `struct complex * t[10];`	tableau de pointeurs vers des structures

`struct complex { float x,y; };`
`struct complex r1, r2, r3;`
`struct complex *t[] = {`
`   &r1, &r2, &r3`
`};`
	tableau de pointeurs vers des structures avec initialisation

`int (*t[10])(int);`	tableau de pointeurs vers des fonctions

`int f1(int a)`
`{`
`/*   corps de la fonction   */`
`}`
 	
`int f2(int a)`
`{`
`/*   corps de la fonction   */`
`}`
 	
`int (*t[2])(int) = {f1, f2};`
	tableau de pointeurs vers des fonctions avec initialisation

E.3 Les pointeurs

`int *p;`	pointeur vers un entier
`int i;` `int *p = &i;`	pointeur vers un entier avec initialisation
`int **p;`	pointeur vers un pointeur vers un entier
`int *q;` `int **p = &q;`	pointeur vers un pointeur vers un entier avec initialisation
`struct complex { float x,y; };` `struct complex *p;`	pointeur vers une structure

`struct complex { float x,y; };`
`struct complex c;`
`struct complex *p = &c;`
	pointeur vers une structure avec initialisation

`int (*p)(int);`	pointeur vers une fonction ayant un paramètre entier et retournant un entier
`int (*p)(int,float);`	pointeur vers une fonction ayant un paramètre entier et un paramètre flottant et retournant un entier
`void (*p)(int);`	pointeur vers une fonction ayant un paramètre entier et ne retournant pas de valeur

`int f(int a)`
`{`
`/*   corps de la fonction   */`
`}`
 	
`int (*p)(int) = f;`
	pointeur vers une fonction ayant un paramètre entier et retournant un entier, avec initialisation

E.4 Les fonctions

`void f(void)`
`{`
`/*   corps de la fonction   */`
`}`
	fonction sans paramètre et sans valeur retournée

`void f(int a)`
`{`
`/*   corps de la fonction   */`
`}`
	fonction avec un paramètre entier, sans valeur retournée

`int f(void)`
`{`
`/*   corps de la fonction   */`
`}`
	fonction sans paramètre retournant un int

`int f(int a, float b)`
`{`
`/*   corps de la fonction   */`
`}`
	fonction avec un paramètre entier et un paramètre flottant, retournant un int

`int f(int a, ...)`
`{`
`/*   corps de la fonction   */`
`}`
	fonction avec un paramètre entier et un nombre variable de paramètres, retournant un int

E.5 Les énumérations

`enum coul {bleu, blanc, rouge};`	`bleu` = 0, `blanc` = 1 et `rouge` = 2
`enum coul {bleu, blanc=10, rouge};`	`bleu` = 0, `blanc` = 10 et `rouge` = 11

`enum coul {`
`   bleu=3, blanc=5, rouge=9`
`};`
	`bleu` = 3, `blanc` = 5 et `rouge` = 9

E.6 Les structures, unions et champs de bits

`struct complex {`
`   float x; /* partie réelle     */`
`   float y; /* partie imaginaire */`
`   };`
	Structure à deux champs flottants. Définit l'étiquette de structure `complex` sans déclarer de variable.

`struct complex {`
`   float x; /* partie réelle     */`
`   float y; /* partie imaginaire */`
`   } c1, c2;`
	Structure à deux champs flottants. Définit l'étiquette de structure `complex` et déclare les deux variables `c1` et `c2`.

`struct {`
`   float x; /* partie réelle     */`
`   float y; /* partie imaginaire */`
`   } c;`
	Structure à deux champs flottants. Déclare la variable `c` sans définir d'étiquette pour la structure.

`enum type {ENTIER, FLOTTANT}`
`struct arith {`
`   enum type typ_val;`
`   union`
`      {`
`      int i;`
`      float f;`
`      } u;`
`   };`
	Union d'un type entier et d'un type flottant associé dans une structure à un indicateur (le champ `typ_val`) permettant de connaitre le type de la valeur stockée dans l'union `u`.

`struct sr {`
`   unsigned int trace : 2;`
`   unsigned int priv : 2;`
`   unsigned int : 1;`
`   unsigned int masque : 3;`
`   unsigned int : 3;`
`   unsigned int extend : 1;`
`   unsigned int negative : 1;`
`   unsigned int zéro : 1;`
`   unsigned int overflow : 1;`
`   unsigned int carry : 1;`
`   };`
	Champs de bits : description du registre d'état du MC68000. Il s'agit d'un mot de 16 bits. Les troisième et cinquième champs ne portent pas de nom.

E.7 Les qualificatifs

Il y a deux qualificatifs : `const` et `volatile` Les exemples présentés sont avec `const`, l'utilisation de `volatile` étant rigoureusement identique.

`const int i;`	Entier constant
`const int *p;`	Pointeur vers un entier constant.
`int * const p;`	Pointeur constant vers un entier.
`const int * const p;`	Pointeur constant vers un entier constant.
`const int t[5];`	Tableau constant : tous les éléments de `t` sont constants.

`struct coord {`
`   int x; int y;`
`   };`
 	
`const struct coord s;`
	Structure constante : tous les champs de `s` sont constants.

Annexe F

Le bêtisier

Cette annexe est une collection des bêtises qu'il faut faire au moins une fois dans sa vie pour être vacciné. La caractéristique de beaucoup de ces erreurs est de ne pas provoquer de message d'erreur du compilateur, rendant ainsi leur détection difficile. La différence entre le texte correct et le texte erroné est souvent seulement d'un seul caractère. La découverte de telles erreurs ne peut donc se faire que par un examen très attentif du source du programe.

F.1 Erreur avec les opérateurs

F.1.1 Erreur sur une comparaison

Ce que voulait le programmeur	Comparer `a` et `b`
Ce qu'il aurait dû écrire	`if (a == b)`
Ce qu'il a écrit	`if (a = b)`
Ce qu'il a obtenu	une affectation de `b` à `a`, suivie d'une comparaison à 0 de la valeur affectée.

Comment est ce possible ?

L'affectation est un opérateur et non pas une instruction.

F.1.2 Erreur sur l'affectation

C'est le pendant de l'erreur précédente.

Ce que voulait le programmeur	Affecter `b` à `a`
Ce qu'il aurait du écrire	`a = b;`
Ce qu'il a écrit	`a == b;`
Ce qu'il a obtenu	La comparaison de `a` à `b`, suivie de l'inutilisation du résultat.

Comment est ce possible ?

Une dérivation possible pour *instruction* est :
instruction :

$\Rightarrow$ *expression*$_{option}$;

Pour que cela ait un sens, il faut que l'*expression* réalise un effet de bord, mais rien ne l'impose dans la définition du langage.

F.2 Erreurs avec les macros

Le mécanisme des macros est réalisé par le préprocesseur qui réalise un traitement en amont du compilateur proprement dit. Le traitement des macros est un pur traitement textuel, sans aucun contexte ; c'est un nid à erreurs.

F.2.1 Un `#define` n'est pas une déclaration

Ce que le programmeur a écrit	Ce qu'il aurait du écrire
`#define MAX 10;`	`#define MAX 10`

Cette erreur peut provoquer ou non une erreur de compilation à l'utilisation de la macro :

- L'utilisation `x = MAX;` aura pour expansion `x = 10;;`, ce qui est licite : il y a une instruction nulle derrière `x = 10;`.

- L'utilisation `int t[MAX];` aura pour expansion `int t[10;];` ce qui génèrera un message d'erreur.

F.2.2 Un `#define` n'est pas une initialisation

Ce que le programmeur a écrit	Ce qu'il aurait du écrire
`#define MAX = 10`	`#define MAX 10`

Cette erreur sera généralement détectée à la compilation, malheureusement le message d'erreur sera émis sur l'utilisation de la macro, et non pas là où réside l'erreur, à savoir la définition de la macro.

F.2.3 Erreur sur macro avec paramètres

La distinction entre macro avec paramètres et macro sans paramètre se fait sur la présence d'une parenthèse ouvrante juste après le nom de la macro, sans aucun blanc entre les deux. Ceci peut amener des résultats surprenant ; comparer les deux exemples suivants :

Définition de la macro	paramètres	corps de la macro
`#define add(a,b) (a + b)`	a et b	`(a + b)`
`#define add (a,b) (a + b)`	aucun	`(a,b) (a + b)`

F.2.4 Erreur avec les effets de bord

Le corps d'une macro peut comporter plusieurs occurrences d'un paramètre. Si à l'utilisation de la macro on réalise un effet de bord sur le paramètre effectif, cet effet de bord sera réalisé plusieurs fois. Exemple :

```
#define CARRE(a) ((a) * (a))
```

l'utilisation de `CARRE(x++)` aura comme expansion `((x++) * (x++))` et l'opérateur `++` sera appliqué deux fois.

F.3 Erreurs avec l'instruction `if`

L'instruction `if` ne comporte ni mot-clé introducteur de la partie *then*, ni terminateur (pas de `fi` dans le style des *if then else fi*). Ceci peut provoquer les erreurs suivantes :

<table>
<tr><th>Ce que le programmeur a écrit</th><th>Ce qu'il aurait du écrire</th></tr>
<tr><td><pre>if (a > b) ;
   a = b;</pre></td><td><pre>if (a > b)
   a = b;</pre></td></tr>
</table>

Le problème vient aussi du fait de l'existence de l'instruction nulle.

<table>
<tr><th>Ce que le programmeur a écrit</th><th>Ce qu'il aurait du écrire</th></tr>
<tr><td><pre>if (a > b)
   if (x > y) x = y;
else
   ...</pre></td><td><pre>if (a > b)
   { if (x > y) x = y; }
else
   ...</pre></td></tr>
</table>

On rappelle qu'un `else` est raccroché au premier `if`.

F.4 Erreurs avec les commentaires

Il y a deux erreurs classiques avec les commentaires :

1. le programmeur oublie la séquence fermante `/*`. Le compilateur "mange" donc tout le texte jusqu'à la séquence fermante du prochain commentaire.

2. On veut enlever (en le mettant en commentaire) un gros bloc d'instructions sans prendre garde au fait qu'il comporte des commentaires. Les commentaires ne pouvant être imbriqués, ça n'aura pas l'effet escompté par le programmeur. La méthode classique pour enlever (tout en le laissant dans le source) un ensemble d'instructions est d'utiliser le préprocesseur :

   ```
   #ifdef NOTDEFINED
   ...
   #endif
   ```

F.5 Erreurs avec les priorités des opérateurs

Les priorités des opérateurs sont parfois surprenantes. Les cas les plus gênants sont les suivants :

– La priorité des opérateurs bit à bit est inférieure à celle des opérateurs de comparaison.

Le programmeur a écrit	il désirait	il a obtenu
`x & 0xff == 0xac`	`(x & 0xff) == 0xac`	`x & (0xff == 0xac)`

– La priorité des opérateurs de décalage est inférieure à celle des opérateurs arithmétiques.

Le programmeur a écrit	il désirait	il a obtenu
`x << 4 + 0xf`	`(x << 4) + 0xf`	`x << (4 + 0xf)`

– La priorité de l'opérateur d'affectation est inférieure à celle des opérateurs de comparaison. Dans la séquence ci-dessous, très souvent utilisée, toutes les parenthèses sont nécessaire :

```
while ((c = getchar()) != EOF)
   {
   ...
   }
```

F.6 Erreur avec l'instruction `switch`

F.6.1 Oubli du `break`

L'instruction de sélection a pour syntaxe : *instruction-sélection* :
⇒ `switch (` *expression* `)` *instruction*
La notion d'alternative de la sélection n'apparaît pas dans la syntaxe : le programmeur doit les réaliser par une liste d'instruction étiquettée par `case` *expression-constante* et terminée par `break`. En cas d'oubli du `break`, une catastrophe s'ensuit.

F.6.2 Erreur sur le `default`

L'alternative à exécuter par défaut est introduite par l'étiquette `default`. Si une faute de frappe est commise sur cette étiquette, l'alternative par défaut ne sera plus reconnue : l'étiquette sera prise pour une étiquette d'instruction sur laquelle ne sera fait aucun `goto`.

```
switch(a)
   {
   case 1 : a = b;
   defult : return(1);       /*   erreur non détectée    */
   }
```

Une version diabolique de cette erreur est relatée dans le livre de Peter Van Der Linden : si la lettre l de `default` est remplacée par le chiffre 1, avec les fontes utilisées pour imprimer les sources, qui verra la différence entre `l` et `1` ?

F.7 Erreur sur les tableaux multidimensionnels

La référence à un tableau `t` à deux dimensions s'écrit `t[i][j]` et non pas `t[i,j]` comme dans d'autres langages de programmation. Malheureusement, si on utilise par erreur la notation `t[i,j]` selon le contexte d'utilisation, elle pourra être acceptée par le compilateur. En effet, dans cette expression la virgule est l'opérateur qui délivre comme résultat l'opérande droit après avoir évalué l'opérande gauche. Comme l'évaluation de l'opérande gauche ne réalise ici aucun effet de bord, cette évaluation est inutile , donc `t[i,j]` est équivalent à `t[j]` qui est l'adresse du sous-tableau correspondant à l'index `j`.

F.8 Erreur avec la compilation séparée

Une erreur classique est d'avoir un tableau défini dans une unité de compilation :

```
int tab[10];
```

et d'utiliser comme déclaration de référence dans une autre unité de compilation :

```
extern int * tab;
```

Rappelons que `int tab[]` et `int *t` ne sont équivalents que dans le seul cas de paramètre formel de fonction. Dans le cas qui nous occupe ici, la déclaration de référence correcte est :

```
extern int tab[];
```

Glossaire

Pour chaque mot de ce glossaire on indique par la notation *Général* si il s'agit d'un concept général dans le domaine des langages de programmation, par *Matériel* si il s'agit d'un concept du domaine de l'architecture des machines, et par *Jargon C* si il s'agit d'un particularisme du langage C. Derrière cette indication, nous donnons le mot anglais utilisé pour désigner ce concept.

adresse *Matériel ; Anglais : address.* La mémoire d'une machine est formée d'une suite d'éléments mémoires. L'adresse d'un élément est son rang dans la suite. *Voir aussi :* élément mémoire.

affectation *Général ; Anglais : assignment.* Opération qui a pour but de stocker une valeur dans la case mémoire correspondant à une variable. L'affectation est généralement réalisée par une instruction, dans le langage C, elle est réalisée par un opérateur.

bloc *Général ; Anglais : block.* Construction d'un langage qui permet de regrouper des déclarations et des instructions.

booléen *Général ; Anglais : boolean.* type dont l'ensemble des valeurs est formé des valeurs *vrai* et *faux*.

chaîne de caractères *Général ; Anglais : string.* Suite contiguë en mémoire de caractères.

champ *Général ; Anglais : field.* Un élément d'un enregistrement.

champ de bits *Jargon C ; Anglais : bit field.* Un champ de structure C dont la taille est donnée en nombre de bits.

compilation séparée *Général ; Anglais : separate compilation.* Technique qui consiste à découper les gros programmes en différentes unités de compilation pour en maîtriser la complexité. Les différentes unités sont compilées séparément, et un éditeur de liens est chargé de transformer les modules objets en programme exécutable.

complément à 2 *Matériel ; Anglais : 2's complement.* Le complément à 2 est une méthode de représentation des nombres signés. Les nombres positifs sont représentés en base 2. Sur 8 bits, la représentation de la valeur 6 est `00000110` et celle de 10 est `00001010`. Pour les nombres négatifs, on part de la valeur absolue que l'on représente en binaire, on la complémente bit à bit et on additionne 1. Sur 8 bits, la représentation de -6 est `11111010` et celle de -10 est `11110110`

déclaration *Général ; Anglais : declaration.* Construction qui associe un identificateur et une entité du langage. Il y a des déclarations de variable, de type, de procédure, de fonction. L'identificateur est un nom pour l'entité déclarée.

durée de vie *Général ; Anglais : lifetime.* Concept qui s'applique aux variables. La durée de vie d'une variable est le temps qui s'écoule entre le moment où on alloue de la mémoire pour cette variable, et le moment où on récupère la mémoire allouée. Il existe classiquement trois types de durée de vie :

- durée de vie du programme : la variable est créée au chargement du programme et détruite à la fin du programme. Une telle variable est dite statique.
- durée de vie d'un bloc : la variable est créée quand on entre dans le bloc et détruite quand on en sort. Une telle variable est qualifiée de dynamique.
- durée de vie programmée : la variable est créée et détruite par des ordres explicites du programmeur. Une telle variable est qualifiée de dynamique.

Voir aussi : pile, tas, statique, dynamique.

effet de bord *Général ; Anglais : side effect.* Modification de l'état de la machine. Un effet de bord peut être interne au programme (par exemple, modification de la valeur d'une variable) ou externe au programme (par exemple écriture dans un fichier). Toute partie de programme qui n'est pas déclarative a pour but soit de calculer une valeur, soit de faire un effet de bord. *Voir aussi :* procédure, fonction.

élément mémoire *Matériel ; Anglais : storage unit.* Une fraction de la mémoire accessible en une seule instruction machine. Une machine comporte généralement au moins trois types d'éléments mémoire : un élément permettant de stocker un caractère, et un élément permettant de stocker un entier, et un élément permettant de stocker un flottant.

enregistrement *Général ; Anglais : record.* Type défini par l'utilisateur permettant de regrouper en une seule entité, des variables de types différents. Au sein de l'enregistrement, les variables sont identifiées par un nom.

ensemble *Général ; Anglais : set.* Le concept mathématique d'ensemble. Certains langages l'offrent (Pascal, Modula-2), d'autres pas (Algol, C, Ada).

évaluation *Général ; Anglais : evaluation.* Ce concept s'applique à une expression. Évaluer une expression c'est calculer sa valeur. *Voir aussi :* évaluation court-circuit.

évaluation court circuit *Général ; Anglais : short circuit evaluation.* Se dit d'une évaluation qui n'évalue pas tous les opérandes. Par exemple, dans le langage C, l'opérateur `&&` évalue d'abord son opérande gauche, si celui-ci est faux, il n'y aura pas d'évaluation de l'opérande droit.

fonction *Général ; Anglais : function.* Possède en commun avec la procédure d'associer un nom à un traitement algorithmique, mais la caractéristique d'une fonction est de délivrer une valeur utilisable dans une expression. Les langages de programmation permettent généralement aux fonctions, en plus du fait de retourner une valeur, de faire un ou plusieurs effets de bord. *Voir aussi :* procédure.

identificateur *Général ; Anglais : identifier.* Synonyme de nom. Dans le domaine des langage de programmation, on parle d'identificateur pour dire nom.

initialisation *Général ; Anglais : initialization.* opération consistant à donner une valeur initiale à une variable. Il y a deux types d'initialisation : les initialisations statiques (à la déclaration de la variable) et les initialisations dynamiques (réalisées par des instructions).

membre *Jargon C ; Anglais : member* Le mot utilisé en C pour dire champ (d'un enregistrement).

mot-clé *Général ; Anglais : keyword.* Un mot dont la signification est fixée par le langage, par opposition aux identificateurs qui sont librement choisis par le programmeur.

objet *Général ; Anglais : object. Voir* source.

paramètre effectif *Général ; Anglais : effective parameter* ou *argument.* La valeur qui est passée en paramètre à la fonction.

paramètre formel *Général ; Anglais : formal parameter* ou *parameter.* Le nom d'un paramètre dans la déclaration d'une procédure ou fonction.

passage de paramètre *Général ; Anglais : parameter passing.* Technique de communication de paramètre entre l'appelant et une procédure ou fonction appelée. Il existe diverses façons de passer les paramètres dont les plus utilisées sont le passage par valeur et le passage par adresse. Le langage C n'utilise que le passage par valeur.

pile *Général ; Anglais : stack.* Dans un langage récursif, les variables locales à une procédure doivent être allouées dans une pile, car à un moment donné il peut y avoir plusieurs activations en cours de la même procédure, et donc plusieurs instanciations des variables de la procédure. *Voir aussi :* durée de vie, tas.

pointeur *Général ; Anglais : pointer.* Type dont les valeurs possibles sont des adresses de variable.

portée *Général ; Anglais : scope.* Ce concept s'applique aux identificateurs. La portée d'un identificateur est la partie du programme où l'ensemble des occurences d'un identificateur font référence à la même déclaration. Ce sont les blocs qui permettent de limiter la portée d'un identificateur.

procédure *Général ; Anglais : procedure.* Permet d'associer un nom à un traitement algorithmique. La procédure est la brique de base de la construction de programme. Le but d'une procédure est de réaliser au moins un effet de bord (sinon elle ne sert à rien).

récursivité *Général ; Anglais : recursivity.* Propriété d'une procédure ou fonction à s'appeler elle-même. Les langages les plus anciens (Fortran, Cobol) ne sont pas récursifs, les langages modernes (Pascal, C) sont récursifs.

source *Général ; Anglais : source.* Forme du programme écrit en langage de programmation. Par opposition à *objet* qui est la forme du programme une fois traduit en instructions machines. Entre la source et l'objet, la sémantique est conservée, seule la forme change.

structure *Jargon C ; Anglais : structure.* Synonyme d'enregistrement.

surchargement *Général ; Anglais : overloading.* Un symbole est dit surchargé quand sa sémantique dépend du contexte. Dans le langage C, le surchargement des opérateurs est une véritable plaie : `*` est à la fois opérateur d'indirection et de multiplication, `&` est à la fois opérateur "adresse de " et "and bit à bit", etc ...

tas *Général ; Anglais : heap.* Zone de la mémoire où sont effectuées les allocations dynamiques de variables explicitement demandées par le programmeur. Dans le langage C, les allocations et les libérations dans le tas se font par des fonctions de la bibliothèque standard : `calloc` et `malloc` pour les allocations, `free` pour les libérations. *Voir aussi :* durée de vie, pile.

type *Général ; Anglais : type.* Attribut d'une variable qui détermine l'ensemble des valeurs que peut prendre cette variable et les opérateurs qu'on peut lui appliquer.

type de base *Général ; Anglais : basic type.* type qui est connu du langage, par opposition aux types définis par le programmeur. Classiquement, les types de base sont *booléen*, *caractère*, *entier*, *flottant*. Le langage C n'a pas le type *booléen*.

type défini par le programmeur *Général ; Anglais : user-defined type.* (On n'a pas inventé de nom pour ce concept : il est désigné par une périphrase.) À partir des types de base, le programeur peut construire de nouveaux types : tableau de *type*, pointeur vers *type*, etc ...

type entier *Jargon C ; Anglais : integral type.* Regroupe les types suivants :

- les `char`
- toutes les variétés de `int` : signés, non signés, longs ou courts.
- les types énumérés (définis par `enum`).

type flottant *Jargon C ; Anglais : floating type.* Regroupe les `float`, `double` et `long double`.

unité de compilation *Général ; Anglais : compilation unit.* Un des fichiers source composant un programme développé selon la technique de la compilation séparée. *Voir aussi :* compilation séparée.

variable *Général ; Anglais : variable.* Une abstraction du concept d'élément mémoire.

variable globale *Général ; Anglais : global variable.* Variable déclarée à l'extérieur de toute procédure ou fonction. Les variables globales sont accessibles par toutes les procédures et fonctions.

variable locale *Général ; Anglais : local variable.* Variable déclarée à l'intérieur d'une procédure ou fonction. Les variables locales sont inaccessibles à l'extérieur de la procédure ou fonction dans laquelle elles sont déclarées.

Index