Introduction au langage C

Bernard Cassagne

Laboratoire de Génie Informatique

IMAG

Grenoble

novembre 1991

Avant-propos

Au sujet de ce manuel

Le but de ce manuel 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 dont la connaissance est nécessaire à la compréhension de ce manuel se trouvent ci-dessous, chaque terme étant accompagné d'une courte définition.

identificateur Sert à nommer les objets du langage.

effet de bord 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.

procédure 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).

fonction 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.

Conventions syntaxiques

Les règles de grammaires suivront 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.
     ex:

     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 réfléter 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` $instruction_2$

   - 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:

     déclareur-init :
     ⇒ *déclareur* $initialiseur_{option}$

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

     déclareur *initialiseur*

     soit en:

     déclareur

Chapter 1

Les bases

1.1 Le compilateur

Les compilateurs C font subir deux transformations aux programmes:

1. un *préprocesseur* fait subir au texte des transformations d'ordre purement lexical,
2. le *compilateur* proprement dit prend le texte généré par le préprocesseur et le traduit en instructions machine.

Le but du préprocesseur est de rendre des services du type traitement de macros et compilation conditionnelle. Initialement, il s'agissait de deux outils différents, mais les possibilités du préprocesseur ont été utilisées de manière tellement extensive par les programmeurs, qu'on en est venu à considérer le préprocesseur comme partie intégrante du compilateur. C'est cette dernière philosophie qui est retenue par l'ANSI dans sa proposition de norme pour le langage C.

1.2 Les types de base

1.2.1 les caractères

Le mot clé désignant les caractères est `char`. `char` est le type dont l'ensemble des valeurs est l'ensemble des **valeurs entières** formant le code des caractères utilisé sur la machine cible. Le type `char` peut donc être considéré comme un sous-ensemble du type entier.

Le code des caractères utilisé 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.

1.2.2 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 donc avoir trois types d'entiers : les `short int`, les `int` et les `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 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 donc avoir deux types d'entiers : les `int` et les `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: les `int`, les `short int`, les `long int` (tous trois signés) et les `unsigned int`, les `unsigned short int` et les `unsigned long int`.

1.2.3 Les flottants

Les flottants peuvent être en simple ou en double précision. Le mot clé désignant les flottants simple précision est `float`, et celui désignant les flottants double précision est `double`. La précision effectivement utilisée pour ces deux types dépend de l'implémentation.

1.3 Les constantes

1.3.1 Les constantes entières

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

Les contantes décimales s'écrivent de la manière usuelle (ex: 372). Les constantes octales doivent commencer par un zéro (ex: 0477). Les constantes hexadécimales doivent commencer par 0x ou 0X et on peut utiliser les lettres majuscules aussi bien que les minuscules pour les chiffres hexadécimaux.

ex:

```
0x5a2b
0X5a2b
0x5A2B
```

Si une constante entière peut tenir sur un `int`, elle a le type `int`, sinon elle a le type `long int`.

On dispose d'une convention permettant d'indiquer au compilateur qu'une constante entière doit avoir le type `long int`. Cette convention consiste à faire suivre la constante de la lettre `l` (en majuscule ou en minuscule). Les notations décimales, octales et hexadécimales peuvent être utilisées avec cette convention.

ex :

```
0L
0456l
0x7affL
```

attention

Ces conventions ne respectent pas les conventions habituelles, puisque `010` devant être interprété en octal, n'est pas égal à `10`.

1.3.2 Les constantes caractères

La valeur d'une constante caractère est la valeur numérique du caractère dans le code de la machine.

Si le caractère dispose d'une représentation imprimable, une constante caractère s'écrit entourée du signe '. ex: `'g'`

En ce qui concerne les caractères ne disposant pas de représentation imprimable, C autorise une notation à l'aide d'un caractère d'*escape* qui est \ :

- Certains d'entres eux, utilisés très fréquemment, disposent d'une notation particulière. Il s'agit des caractères suivant:

new line	`'\n'`
horizontal tabulation	`'\t'`
back space	`'\b'`
carriage return	`'\r'`
form feed	`'\f'`
back slash	`'\\'`
single quote	`'\{''}`

- les autres disposent de la notation $\backslash x$ où x est un nombre exprimé en octal. ex:

`'\001'`	SOH
`'\002'`	STX
`'\003'`	ETX
etc...	

1.3.3 Les constantes flottantes

La notation utilisée est la notation classique par mantisse et exposant. Pour introduire l'exposant, on peut utiliser la lettre `e` sous forme minuscule ou majuscule. ex:

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}

Toute constante flottante est considérée comme étant de type flottant double précision.

1.4 Les chaînes de caractères littérales

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

```
"ceci est une chaine"
```

Toutes les notations de caractères non imprimables définies en 1.3.2 sont utilisables dans les chaînes. ex:

```
"si on m'imprime,\nje serai sur\ntrois lignes a cause des deux new line"
```

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.

ex:

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

Le compilateur rajoute à la fin de chaque chaîne un caractère mis à zéro. (Le caractère dont la valeur est zéro est appellé *null* dans le code ASCII). Cette convention de fin de chaîne est utilisée par les fonctions de la librairie standard.

1.5 constantes nommées

Il n'y a pas en C de possibilité de donner un nom à une constante. On peut cependant réaliser un effet équivalent grâce au préprocesseur. Lorsque le préprocesseur lit une ligne du type:

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

il remplace dans toute la suite du source, toute nouvelle occurence 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`.

1.6 Déclarations de variables ayant un type de base

Une telle déclaration se fait en faisant suivre le nom du type, par la liste des noms des variables.

ex :

```
int i;
int i,j;
short int k;
float f;
double d1,d2;
```

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

ex:

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

1.7 Les opérateurs les plus usuels

1.7.1 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, 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 a deux effets:

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

  ex :

```
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`.

- Restrictions:

  La *lvalue* doit désigner une variable ayant un type de base. Il n'est donc pas possible d'affecter un tableau à un tableau, par exemple.

- 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 à un flottant, la valeur d'une expression entière.

1.7.2 L'addition

- Syntaxe:

  expression :
  ⇒ $expression_1$ + $expression_2$

- Sémantique:

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

  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 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 réel et d'une expression délivrant un entier.

1.7.3 La soustraction

- Syntaxe:

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

  expression :
  ⇒ - *expression*
  ⇒ $expression_1$ - $expression_2$

- 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.

1.7.4 La multiplication

- Syntaxe:

  expression :
  ⇒ $expression_1$ * $expression_2$

- 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.

1.7.5 La division

- Syntaxe:

  expression :
  ⇒ $expression_1$ `/` $expression_2$

- Sémantique:

  Contrairement a 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 réelle.

  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 réelle, il s'agit d'une division réelle.

  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 zéro). ex: `13 / 2` délivre la valeur 6 et `-13 / 2` ou `13 / -2` peuvent délivrer -6 ou -7 mais le résultat sera généralement -6.

  Dans le cas de la division réelle, les remarques concernant l'ordre d'évaluation des opérandes ainsi que les éventuelles conversions de type faites au sujet de l'addition s'appliquent également.

1.7.6 L'opérateur modulo

- Syntaxe:

  expression :
  ⇒ $expression_1$ `%` $expression_2$

- 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. ex : `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` doit avoir pour valeur a.

1.7.7 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.

1.8 Les instructions les plus usuelles

1.8.1 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.

  ex:

```
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.

1.8.2 Instruction composée

- Syntaxe:

  instruction :
  ⇒ `{`
  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

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

1.8.3 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 une possible ambiguité sur l'instruction `if` dont dépend une partie `else`, l'ambiguité est levée en faisant dépendre le `else` de l'instruction `if` la plus proche.

   Par exemple, dans le cas de :

   ```
   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

Etant 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

ex:

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

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

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

Chapter 2

Fonctions et procédures

2.1 Définition d'une fonction

- Syntaxe:

  définition-de-fonction :

  ⇒ *type* *identificateur* (*liste-d'identificateurs*$_{option}$)
  liste-de-déclarations$_{1\ option}$
  {
  liste-de-déclarations$_{2\ option}$
  liste-d'instructions
  }

- Sémantique:

  type est le type de la valeur rendue par la fonction, *identificateur* est le nom de la fonction, *liste-d'identificateurs* est la liste des noms des paramètres formels, et la *liste-de-déclarations*$_1$ est la liste des déclarations des paramètres formels permettant d'indiquer le type de ces paramètres. La *liste-de-déclarations*$_2$ 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.*

  ex:

```
int sum_square(i,j)   /*   la fonction sum_square delivre un int     */
int i,j;              /*   declaration des parametres formels        */
{
int resultat;         /*   declaration des variables locales         */
```

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

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.

ex:

```
int max(i,j)           /*   la fonction max delivre un int       */
int i,j;               /*   declaration des parametres formels   */
{                      /*   pas de variables locales pour max    */

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

Dans le cas où la dernière instruction exécutée par une fonction n'est pas une instruction **return,** la valeur rendue par la fonction est indéterminée.

La liste des paramètres formels est optionnelle de façon à permettre l'écriture de fonctions sans paramètres.

ex:

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

D'autre part, la *liste-de-déclarations*$_1$ est optionnelle, elle est donc omise quand il n'y a pas de paramètre formel, mais on peut quand même l'omettre quand il y a des paramètres formels ! Dans ce cas, le type des paramètres formels est pris par défaut comme étant **int**. Par exemple, la fonction **max** aurait pu être programmée de la façon suivante:

```
int max(i,j)                          /*   la fonction max delivre un int   */
{                                     /*   pas de declaration pour i et j   */
if (i > j) return(i); else return(j);
}
```

On peut considérer que profiter de cette possibilité est un mauvais style de programmation.

2.2 Appel d'une fonction

- Syntaxe:

  expression :
  $\Rightarrow$ *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 activée. La valeur rendue par la fonction est la valeur de l'expression appel de fonction.

ex:

```
{
int a,b,c;
double d;

d = sum_square(a,b) / 2;      /*   appel de sum_square   */
c = max(a,b);                 /*   appel de max          */
}
```

2.3 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", il existe un type spécial du langage qui porte le nom de `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.

```
void print_add(i,j)       /*   la procedure et ses parametres formels    */
int i,j;                  /*   declaration des parametres formels        */
{
int r;                    /*   une variable locale a print_add           */

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

void prints()             /*   une procedure sans parametres             */
{
int a,b;                  /*   variables locales a prints                */

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

2.4 Omission du type retourné par une fonction

Nous avons vu que la syntaxe de définition d'une fonction est:
définition-de-fonction :

⇒ *type* *identificateur* (*liste-d'identificateurs*$_{option}$)
liste-de-déclarations$_{1\ option}$
{
liste-de-déclarations$_{2\ option}$
liste-d'instructions
}

En réalité *type* est optionnel. Si le programmeur omet d'indiquer quel est le type de la valeur rendue par la fonction, le compilateur prend par défaut le type `int`.

1. si le programmeur veut écrire une fonction rendant un `int`, profiter du fait que le compilateur prendra la valeur `int` par défaut pour omettre *type*, peut être considéré comme étant un mauvais style de programmation.

2. si le programmeur veut écrire une procédure, omettre *type* plutôt que d'indiquer `void` est à la rigueur acceptable. On doit d'ailleur remarquer que le type `void` n'existait pas dans la première définition du langage, et qu'il existe donc des masses de code où les procédures sont codées de cette façon.

On peut donc à la rigueur écrire les procédures de la manière suivante:

```
print_add(i,j)            /*   pas d'indication du type de valeur rendue */
int i,j;
{
int r;

r = i + j;
...
}

prints()                 /*   pas d'indication du type de valeur rendue */
{
int a,b;

a = 12; b = 45;
print_add(a,b);
print_add(13,67);
}
```

2.5 Impression formattée

Il existe une procédure standard permettant de réaliser des sorties formattées: il s'agit de la procédure `printf`. On l'appelle de la manière suivante:

`printf` (*chaine-de-caractères* , *liste-d'expressions*) ;

chaine-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.

ex:

```
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 (*chaine-de-caractères*) ;
par exemple, pour imprimer le mot erreur en le soulignant, on peut écrire:

```
printf("e\b_r\b_r\b_e\b_u\b_r\b_");   /*   \b est back-space   */
```

2.6 Structure d'un programme

Nous ne considèrerons pour débuter que le cas de programmes formés d'un seul module source.

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 :
⇒ *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**.

ex:

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

void p1(k)          /*   debut de la definition de la procedure p1      */
int k;              /*   p1 a un seul parametre entier : k              */
{
```

```
int s,t;            /*   variables locales a p1                          */

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

int f1(x,y)         /*   debut de la definition de la fonction f1        */
double x,y;         /*   f1 a deux parametres reeels: x et y             */
{
double u,v;         /*   variables locales a f1                          */

...                 /*   instructions de f1 qui peuvent acceder a        */
...                 /*   x,y,i,j,a,b,c,u et v                            */
}                   /*   fin de la definition de f1                      */

void main()         /*   debut du main, il n'a pas de parametre          */
{
...                 /*   instructions du main qui appellerons p1 et f1   */
}                   /*   fin de la definition de main                    */
```

2.7 Mise en œuvre du compilateur C sous UNIX

Le lecteur sera supposé maitriser 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 le compiler sous UNIX 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.

ex:

```
void main()
{
printf("ca marche!!\n");
}
```

2.8 Exercice

Ecrire un programme comportant:

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

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.

```
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 a une certaine valeur                                    */
/*                                                                           */
/*   Interface:                                                              */
/*      h, m, s sont les valeurs a donner a heures, minutes, secondes        */
/*                                                                           */
/*****************************************************************************/
void set_heure(h,m,s)
int h,m,s;

{
heures = h; minutes = m; secondes = s;
}

/*****************************************************************************/
/*                                                                           */
/*                              tick                                         */
/*                                                                           */
/*   But:                                                                    */
/*      Incremente 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                                         */
/*                                                                           */
/*****************************************************************************/
void 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();
}
```

Chapter 3

Les tableaux

Dans ce chapitre nous allons voir tout d'abord 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 certains nombre d'opérateurs très utiles pour mettre en œuvre ces instructions.

3.1 Les tableaux

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

Une déclaration de tableau dont les éléments ont un type de base, a une structure très proche d'une déclaration de variable ayant un type de base. La seule différence consiste à indiquer entre crochets le nombre d'éléments du tableau après le nom de la variable.

ex:

```
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.

ex:

```
#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.

3.1.2 Initialisation d'un tableau

Lorsqu'un tableau est externe à toute fonction, il est possible de l'initialiser avec une liste d'expressions constantes séparées par des virgules, et entourée des signes { et }.

ex:

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

Il est possible de 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.

ex:

```
#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'initialiseur, soit initialiser le tableau par des instructions.

Cas particulier des tableaux de caractères

Un tableau de caractères peut être initialisé selon la même technique. On peut écrire par exemple:

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

Comme cette méthode est extrêmement lourde, le langage C a prévu la possibilité d'initialiser un tableau de caractères à l'aide d'une chaine littérale. Par exemple:

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

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

Il est admissible que la taille déclarée pour le tableau soit supérieure à la taille de la chaine littérale.

ex:

```
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 chaine littérale et de donner cette taille au tableau.

ex:

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

3.1.3 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 :
  ⇒ *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.

ex:
Dans le contexte de la déclaration:

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

on peut écrire:

```
x = t[i];       /*  reference a l'element d'indice i du tableau t       */
t[i+j] = k;     /*  affectation de l'element d'indice i+j du tableau t  */
```

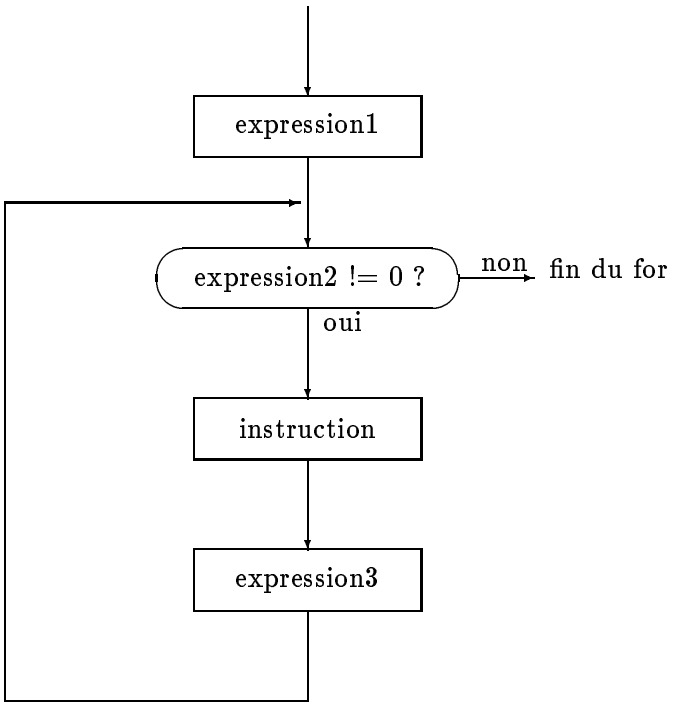
3.2 Les instructions itératives

3.2.1 Instruction `for`

- Syntaxe:

  instruction :
  ⇒ `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. $expresion_2$ est elle utilisée pour le test de bouclage. *instruction* est le travail de la boucle.

Exemple de boucle `for`

initialisation d'un tableau

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

3.2.2 Instruction `while`

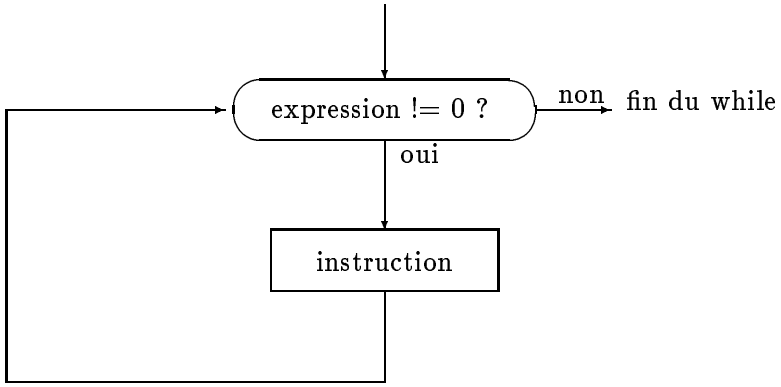
- Syntaxe:

  instruction :

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

- Sémantique:

  l'exécution réalisée correspond à l'organigramme suivant:



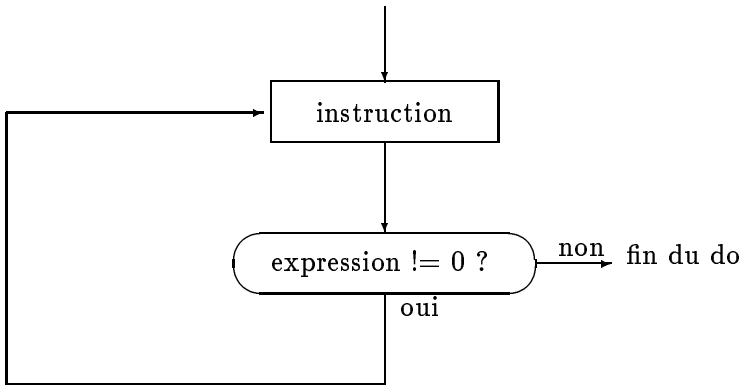
3.2.3 Instruction `do`

- Syntaxe:

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

- Sémantique:

  l'exécution réalisée correspond à l'organigramme suivant:



3.2.4 Instruction `break`

- Syntaxe:

  instruction :
  ⇒ `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;
```

3.2.5 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` à la recherche d'un élément satisfaisant une certaine condition algorithmiquement complexe à écrire, mais que l'on sache qu'une valeur négative ne peut pas convenir:

```
for (i = 0; i < N; i = i + 1)
   {
   if (t[i] < 0 ) continue;   /*   on passe au i suivant dans le for   */
   ...                        /*   algorithme de choix                 */
   }
```

3.3 Les opérateurs

3.3.1 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 a zero l'element d'indice 0  */
t[i++] = 0;  /*  met a zero l'element d'indice 1  */

i = 1;
```

```
t[++i] = 0;  /*  met a zero l'element d'indice 2  */
t[++i] = 0;  /*  met a zero l'element d'indice 3  */
```

3.3.2 Opérateur pré et postdécrement

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

1. il peut s'utiliser en préfixe ou en postfixé,
2. il s'applique à une *lvalue*,
3. 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 a zero l'element d'indice 9  */
t[i--] = 0;  /*  met a zero l'element d'indice 8  */

i = 8;
t[--i] = 0;  /*  met a zero l'element d'indice 7  */
t[--i] = 0;  /*  met a zero l'element d'indice 6  */
```

3.3.3 Quelques utilisations typiques de ces opérateurs

Utilisation dans les instructions *expression*

On a vu qu'une des formes d'instruction possibles en C est:
expresssion ;
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++;    /*   incrementation de i   */
j--;    /*   decrementation 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++)
   {
   ...
   }
```

3.3.4 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 chainés.

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
```

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 satisfaisant le test `t[i] == 234`, il va arriver un moment où on va évaluer `i < n && t[i] == 234` avec `i > n` et la sémantique de l'opérateur `&&` assure que l'expression `t[i] == 234` ne sera pas évaluée. On ne cours donc pas le risque d'avoir une erreur matérielle dans la mesure où t[i+1] 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);
```

3.3.5 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);
```

3.3.6 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.

3.4 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).

Ecrire 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

Ecrire 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`.

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

Chapter 4

Les pointeurs

4.1 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.

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

La déclaration d'une variable pointeur vers un type de base a une structure très proche de celle de la déclaration d'une variable ayant un type de base. La seule différence consiste à faire précéder le nom de la variable du signe `*`.

ex:

```
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 precision   */
char *pc;         /*   pc pointeur vers un char                        */
```

4.3 Opérateur adresse de

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

```
int i;
```

```
int *pi;

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

4.4 Opérateur d'indirection

Lorsque l'opérateur `*` est utilisé en opérateur préfixé, il ne s'agit pas de l'opérateur de multiplication, mais de l'opérateur indirection, qui, appliqué à une valeur de type pointeur, désigne la variable pointée.

On peut écrire par exemple:

```
int i;
int *pi;

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

4.5 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.

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

i = 1;
p = &i;

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

4.6 Pointeurs et opérateurs additifs

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 le i^{eme} élément du tableau, p + j **est une valeur de type pointeur vers T**, qui repère le $(i + j)^{eme}$ élément du tableau (en supposant qu'il existe).

Il en va de même avec l'opérateur soustraction, et si p repère le i^{eme} élément d'un tableau, p - j repère le $(i - j)^{eme}$ élément du tableau (toujours en supposant qu'il existe).

ex:

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

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

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

4.7 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** `int` qui est égale à j - i.

4.8 Exercice

Déclarer et initialiser statiquement un tableau d'entiers `t` avec des valeurs dont certaines seront nulles.

Ecrire une procédure main 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.

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

/*****************************************************************************/
/*                                                                           */
/*                              main                                         */
/*                                                                           */
/*****************************************************************************/
void main()
{
int *pdeb,*pfin,*p;

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

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

4.9 Passage de paramètres

4.9.1 Les besoins du programmeur

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

- soit il désire passer à la procédure 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)).
- 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`.

4.9.2 Comment les langages de programmation satisfont ces besoins

Face à ces besoins, les concepteurs de langages de programmation ont imaginés 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 statégie 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 statégie choisie par les concepteurs du langage C.

4.9.3 La statégie 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(a,b,c)
int a,b;
int *c;    /*   c repere l'entier ou on veut mettre le resultat   */

{
*c = a + b;
}

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

/*   on passe l'adresse de k a add comme troisieme parametre   */
add(i,j,&k);
}
```

4.10 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 fait `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.

4.11 Une dernière précision

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

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

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

C'est la seconde solution qui a étée 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(n)
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`.

4.12 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 ponctation 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.

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

```
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 caractere passe en parametre                               */
/*                                                                           */
/*   Interface:                                                              */
/*      p : pointe le caractere a crypter                                    */
/*                                                                           */
/*****************************************************************************/
void crypt(p)
char *p;

{
#define HAUT 1
#define BAS 0
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                                         */
/*                                                                           */
/*****************************************************************************/
void main()
{
char *p;
int i;

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

/*   impression du resultat   */
printf("resultat:\n");
i = 0;
while (mess[i]) printf("%c",mess[i++]);
}
```

4.13 Lecture formattée

Il existe une fonction standard de lecture formattée qui fonctionne selon le même principe que la procédure `printf`.

Sa syntaxe d'utilisation est la suivante:

`scanf` (*chaine-de-caractères* , *liste-d'expressions*) ;

la *chaine-de-caractères* indique sous forme de séquences d'échappement le format des 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.

ex:

```
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`.

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[1] sur l'éxécution 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.

4.14 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.

4.14.1 Instruction `switch`

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

[1] Le terme avorter est à prendre au sens technique de *abort*

- Syntaxe:

  instruction :

  ⇒ `switch (` *expression* `)`
  `{`
  `case` $expression_1$: $liste\text{-}d'instructions_{1\ option}$ `break;`$_{option}$
  `case` $expression_2$: $liste\text{-}d'instructions_{2\ option}$ `break;`$_{option}$

  `....`

  `case` $expression_n$: $liste\text{-}d'instructions_{n\ option}$ `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 *liste-d'instructions* correspondante 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 exite, 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:

```
#define BLEU 1
#define BLANC 2
#define ROUGE 3

void print_color(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 numero 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 pourra 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:

```
#define POSSIBLE 0
#define IMPOSSIBLE 1
void print_cas(cas)
int cas;
{
switch (cas)
   {
   case IMPOSSIBLE: printf("im");
   case POSSIBLE:   printf("possible\n"); break;
   case default:    printf("erreur interne du logiciel numero 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;   */
```

4.14.2 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 étiquettée par *identificateur*.

ex:

```
{
etiq2:
...             /*   des instructions                       */
goto etiq1;     /*   goto avant definition de l'etiquette   */
...             /*   des instructions                       */
etiq1:
...             /*   des instructions                       */
goto etiq2;     /*   goto apres definition de l'etiquette   */
}
```

4.14.3 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+1, et ensuite exécute une fois `t[i] = i` ce qui a probablement pour effet d'écraser la variable déclarée juste après le tableau `t`.

4.15 Exercice

Ecrire 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

```
#define VRAI 1
#define FAUX 0

/*****************************************************************************/
/*                                                                           */
/*                              main                                         */
/*                                                                           */
/*****************************************************************************/
void main()
{
int i,j,r;  /*   les operandes  */
char c;     /*   l'operateur   */
char imp;   /*   booleen de demande d'impression du resultat  */

while (1)
   {
   scanf("%d %c %d",&i,&c,&j);
   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 zero\n");
            imp = FAUX;
            }
         else r = i / j;
         break;
      case '%' : r = i % j; break;
      default  : printf("l'operateur %c est incorrect\n",c); imp = FAUX;
      }   /*   fin du switch   */

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

Chapter 5

Tableaux et pointeurs

5.1 Retour sur les tableaux

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

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

et qu'on disposait d'un opérateur d'indexation, noté `[]`, permettant d'obtenir un élément du tableau. L'élément d'index `i` du tableau `t` se désignant par `t[i]`.

Ceci est suffisant pour les utilisations les plus simples des tableaux, mais dès que l'on veut faire des choses plus complexes, (par exemple passer des tableaux en paramètre), il est nécessaire d'aller plus à fond dans le concept de tableau tel que l'envisage le langage C.

De manière à bien mettre en évidence les particularités des tableaux dans le langage C, nous allons faire un rappel sur la notion de tableau telle qu'elle est généralement envisagée dans les langages de programmation.

5.2 La notion de tableau en programmation

La plupart des langages de programmation disposent du concept de tableau. Une telle notion offre au programmeur deux services principaux:

- déclarer des tableaux
- d'utiliser un opérateur d'indexation, qui peut être noté `()` ou `[]`, admettant en opérande un tableau `t` et un entier `i`, et délivrant l'élément d'index `i` de `t`.

Derrière les particulariés de chaque langage (parfois les bornes des tableaux peuvent être connue dynamiquement, parfois les bornes doivent être connues statiquement, etc...) le programmeur peut cependant se raccrocher à un certain nombre de points communs.

Généralement en effet, quand on déclare un tableau de nom `t`,

1. `t` est une variable,
2. `t` est de type tableau de *quelquechose*,
3. `t` désigne le tableau en entier.

Voyons ce qu'il en est en ce qui concerne la façon dont C envisage les tableaux.

5.3 La notion de tableau dans le langage C

Sous une similitude de surface, le langage C cache de profondes différences avec ce que nous venons de voir. En effet, lorsqu'en C, on déclare un tableau de nom `t`,

1. `t` **n'est pas** une variable,
2. `t` **n'est pas** de type tableau de *quelquechose*,
3. `t` **ne désigne pas** le tableau en entier.

Que se passe-t-il alors?
En C, quand on écrit:

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

on a bien déclaré une variable de type tableau, mais cette variable **n'a pas de nom**.

Le nom `t` est le nom d'une **constante** qui est de type pointeur vers `int`, et dont la valeur est l'adresse du premier élément du tableau. (En fait, `t` est rigoureusement identique à `&t[0]`).

Le fait de déclarer une variable qui n'a pas de nom est un mécanisme qui est bien connu dans les langages de programmation. La situation est comparable à ce que l'on obtient lorsqu'on fait l'allocation dynamique d'une variable (via le new PASCAL, l'allocate de PL/1 etc...). La variable allouée n'a pas de nom, elle ne peut être désignée que via un pointeur. Ce qui est original ici, c'est de retrouver ce mécanisme sur une allocation de variable qui est une allocation statique.

Une telle façon d'envisager les tableaux dans le langage C va avoir deux conséquences importantes: la première concerne l'opérateur d'indexation et la seconde la passage de tableaux en paramètre.

5.3.1 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'indirection 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`.

Si on est allergique au fait d'avoir des tableaux dont les bornes vont de 0 à n-1 et si on préfère avoir des bornes de 1 à n, on peut utiliser la méthode suivante:

```
int t[10];
int * p;
...
p = &t[-1];
```

Le "tableau" `p` est un synonyme du tableau `t` avec des bornes allant de 1 à 10.

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.

5.3.2 Passage de tableau en paramètre

Nous avons vu que le nom d'un tableau est en fait l'adresse du premier élément. Lorsqu'un tableau est passé en paramètre effectif, c'est donc cette adresse qui sera 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(t,nb_elem)   /*  definition de imp_tab   */
int *t;
int nb_elem;

{
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 les deux lignes

```
void imp_tab(t)
int *t;
```

il n'est pas clair 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 au niveau de l'en-tête de la procédure, le langage C admet que l'on puisse déclarer un paramètre formel de la façon suivante:

```
void proc(t)
int t[];

{
...   /*   corps de la procedure proc   */
}
```

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

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(t,nb_elem)   /*  definition de imp_tab   */
int t[];
int nb_elem;

{
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];

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

5.4 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 fasse par valeur.

Bien sûr il n'en est rien. Puisqu'en C le nom d'un tableau a pour valeur l'adresse du premier élément du tableau, c'est cette valeur qui est passée en paramètre. C'est donc bien encore du passage par valeur.

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 elements du tableau t  */
void incr_tab(t,nb_elem)
int t[];
int nb_elem;

{
int i;

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

5.5 Exercice

1. Déclarer et initialiser deux tableaux de caractères (ch1 et ch2).
2. Ecrire 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. Ecrire 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`.

```
#define NULL_C '\0'

char ch1[] = "cette chaine comporte 35 caracteres";
char ch2[] = "et celle ci fait 30 caracteres";

/*****************************************************************************/
/*                                                                           */
/*                              lg_chaine1                                   */
/*                                                                           */
/*   But:                                                                    */
/*      calcule la longueur d'une chaine de caracteres                       */
/*                                                                           */
/*   Interface:                                                              */
/*      ch : la chaine de caracteres                                         */
/*      valeur rendue : la longueur de ch                                    */
/*                                                                           */
/*****************************************************************************/
int lg_chaine1(ch)
char ch[];

{
int i = 0;

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

return(i);
}

/*****************************************************************************/
/*                                                                           */
/*                              lg_chaine2                                   */
/*                                                                           */
/*   But:                                                                    */
/*      identique a celui de lg_chaine1                                      */
/*                                                                           */
/*****************************************************************************/
int lg_chaine2(ch)
char *ch;

{
int i = 0;

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

return(i);
}

/*****************************************************************************/
/*                                                                           */
/*                              main                                         */
/*                                                                           */
```

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

5.6 Déclaration de tableaux multi-dimensionnels

En C, un tableau multi-dimensionnel est considéré comme étant un tableau dont les éléments sont eux mêmes des tableaux.

Un tableau à deux dimensions se déclarera donc de la manière suivante:

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

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

1. t est une constante de type pointeur vers un tableau de 20 int.
2. sa valeur est l'adresse d'une zone suffisante pour stocker un tableau de 10 tableaux de 20 int.

5.7 Accès aux éléments d'un tableau multi-dimensionnel

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

5.8 Passage de tableaux multi-dimensionnels 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(t)
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 donc besoin de connaitre N.

On a vu qu'il était possible de passer en paramètre à une procédure la taille d'un tableau unidimensionnel, de façon à permettre à cette procédure d'accepter des paramètres effectifs qui aient des tailles différentes.

En ce qui concerne les tableaux à plusieurs dimensions, ceci reste vrai pour la première dimension, mais est faux pour les autres.

ex:

```
#define P 10

void raz_mat(t,n)
int t[][P];
```

```
int n;       /*   taille de la  dimension   */

{
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.

5.9 Initialisation d'un tableau multi-dimensionnel

Lorsqu'un tableau multi-dimensionnel est déclaré à l'extérieur de toute procédure ou fonction, il peut être initialisé à la déclaration.

ex:

```
int t[5][5] = {
   { 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}
};
```

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

5.10 Exercice

1. Déclarer et initialiser statiquement une matrice [5,5] d'entiers (`tab`)
2. Ecrire 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`.

```
#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(t)
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                                         */
/*                                                                           */
/*****************************************************************************/
main()
{
print_mat(tab);
}
```

Chapter 6

Les entrées sorties

A 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 nous restreignant à des possibilités communes à UNIX System V et UNIX Berkeley, de manière à n'avoir aucun problème de portabilité.

L'usage de toutes les fonctions présentées nécessitant d'utiliser les facilités d'inclusion de source du préprocesseur, nous allons commencer par cela.

6.1 Inclusion de source

Lorsqu'on développe un gros logiciel, il est généralement nécessaire de le découper en modules de taille raisonnable.

Un fois un tel découpage réalisé, il est courant que plusieurs modules aient certaines parties communes (par exemple des définitions de constantes à l'aide de directives `#define`).

De facon à éviter la répétition de ces parties communes dans les modules, 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*.

L'endroit ou se fait la recherche de *nom-de-fichier* dépend du système et des caractères choisis pour entourer *nom-de-fichier* (" ou < et >).

La philosophie est la suivante:

- si on utilise ", le fichier à inclure est un fichier utilisateur, il sera donc recherché dans l'espace de fichiers de l'utilisateur (sous UNIX ce sera le directory courant).

- si on utilise < et >, le fichier est un fichier système, il sera recherché dans un ou plusieurs espaces de fichiers systèmes (sous UNIX ce sera `/usr/lib/include`).

6.2 Ouverture et fermeture de fichiers

6.2.1 Ouverture d'un fichier : `fopen`

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

Paramètres

Cette fonction admet deux paramètres qui sont respectivement:

- une chaîne de caractères qui est le nom du fichier auquel on veut accéder.
- une chaîne de caractères qui est le mode de l'ouverture, et qui peut prendre les valeurs suivantes:

  `"r"` ouverture en lecture.

  `"w"` ouverture en écriture.

  `"a"` ouverture en écriture à la fin.

  Si on ouvre un fichier qui n'existe pas en `"w"` ou en `"a"`, il est crée.

  Si on ouvre en `"w"` un fichier qui existe deja, son ancien contenu est perdu.

  Si on ouvre un fichier qui n'existe pas en `"r"`, c'est une erreur.

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 avéré impossible, `fopen` rend la valeur `NULL`, constante prédéfinie dans `stdio.h`

Utilisation typique de `fopen`

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

if ((fp = fopen("donnees","r")) == NULL)
   printf("Impossible d'ouvrir le fichier donnees\n");
```

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`.

6.2.2 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` qui admet comme paramètre une valeur de type pointeur vers `FILE` qui repère le fichier à fermer.

Utilisation typique:

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

fclose(f);
```

6.3 Lecture et écriture par caractère sur fichier

6.3.1 lecture par caractère: `getc`

`getc` est une fonction à un seul paramètre qui est une valeur de type pointeur vers `FILE`. Elle lit un caractère du fichier et le retourne **dans un int**. La valeur rendue est un `int` car elle peut prendre en plus de toutes les valeurs possibles pour les caractères, la valeur `EOF` qui est une constante prédéfinie se trouvant dans `stdio.h`. Cette valeur indique que l'on a atteint la fin du fichier.

Utilisation typique:

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

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

6.3.2 écriture par caractère : `putc`

`putc` admet deux paramètres qui sont respectivement :

1. le caractère à écrire dans le fichier
2. la valeur de type pointeur vers `FILE` qui repère le fichier.

En ce qui concerne le premier paramètre, on peut passer une valeur de type `char`, mais aussi une valeur de type `int`. Dans ce dernier cas, la fraction de l'entier qui est interprété comme un `char` est celle dans laquelle `getc` met la valeur du caractère qu'il lit.

Ce qui fait que l'on peut écrire:

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

```
while ((c = getc(fi)) != EOF)
   putc(c,fo);
```

aussi bien que:

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

while ((resu = getc(fi)) != EOF)
   {
   c = resu;
   putc(c,fo);
   }
```

6.4 Lecture et écriture par lignes sur fichier

6.4.1 lecture par ligne : `fgets`

`fgets` admet trois paramètres :

1. un tableau de caractères
2. un `int` donnant la taille du tableau.
3. un pointeur vers `FILE` caractérisant le fichier.

`fgets` lit les caractères du fichiers et les range dans le tableau jusqu'à rencontre d'un *line-feed* (qui est mis dans le tableau) 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 à zéro (un *null).*

`fgets` rend un pointeur vers le tableau de caractères, ou `NULL` sur fin de fichier.

Utilisation typique:

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

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

6.4.2 écriture par chaîne : `fputs`

fputs admet deux paramètres :

1. un tableau de caractères (terminé par un caractère *null*)

2. un pointeur vers `FILE` caractérisant le fichier.

`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 indifférement à écrire une ligne ou une chaîne quelconque.

Utilisation typique:

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

fputs(ligne,fo);
```

6.5 E/S formattées sur fichiers

6.5.1 Ecriture formattée: `fprintf`

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

`fprintf` (*file-ptr* , *format* , $param_1$, $param_2$, ... , $param_n$)

file-ptr est une valeur de type pointeur vers `FILE` qui repère le fichier sur lequel on veut écrire. *format* est une chaîne de caractères qui sera recopiée sur le fichier.

En plus des caractères à copier tels quels, *format* contient des séquences d'escape décrivant la manière dont doivent être écrits les paramètres $param_1$, $param_2$, ... $param_n$. Le caractère introduisant une séquence d'escape est `%`. Une séquence d'escape se compose des éléments suivants:

- 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 les formats `c`, `d`, `s`, `u` cet indicateur ne change rien. 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. Pour les formats `g` et `G`, cet indicateur empêche les zéros de la fin d'être enlevés.

- Une chaîne de nombres décimaux indiquants 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 gauche (ou à droite si l'indicateur - a été utilisé). Le caractère qui sert à compléter $param_i$ est le caractère blanc ou le caractère zéro si cette taille minimum du champ d'impression commence par un zéro.

- Le caractère . (point) suivi d'une chaîne de caractères décimaux indiquant la précision avec laquelle $param_i$ doit être imprimé. Cette précision est le nombre de caractères d'impression de $param_i$ pour les formats `d`, `o`, `u`, `x` et `X`, le nombre de chiffres après la virgule pour les formats `e`, `E` et `f`, et le nombre maximum de chiffres significatifs pour les formats `g` et `G`, et le nombre maximum de caractères pour le format `s`.

La chaîne de chiffres décimaux indiquant la taille maximum du champ d'impression et/ou la chaîne de chiffres décimaux 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 $param_{i-1}$.

- Le caractère `l` qui, s'il est utilisé avec l'un des formats `d`, `o`, `u`, `x` ou `X`, signifie que $param_i$ sera interprété comme un long. Dans le cas où `l` est utilisé avec un autre format, il est ignoré.

- un caractère qui peut prendre les valeurs suivantes :

  `d` $param_i$ sera écrit en décimal signé.

  `u` $param_i$ sera écrit en décimal non signé.

  `o` $param_i$ sera écrit en octal non signé.

  `x,X` $param_i$ sera écrit en hexadécimal non signé. La notation hexadécimale utilisera les lettres `abcdef` dans le cas du format `x`, et les lettres `ABCDEF` dans le cas du format `X`.

  Dans les 4 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.

  `c` $param_i$ sera interprété comme un caractère

  `s` $param_i$ sera interprété comme l'adresse d'une chaîne de caractères terminée par *null*. Cette chaîne sera imprimée.

  `e,E` $param_i$ sera interprété comme un flottant 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'apparait pas. Dans le cas du format `E`, la lettre `E` est imprimée à la place de `e`.

`f` *param*$_i$ sera interprété comme un flottant 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'apparait pas.

`g,G` *param*$_i$ sera interprété comme un flottant 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 que la précision, il sera imprimé sous le format `e`, sinon il sera imprimé sous le format `f`. Dans ce qui précède, l'utilisation du format `G` implique l'utilisation du format `E` à la place du format `e`.

Pour mettre le caractère **%** dans standard output, il faut écrire **%%** dans le format.

6.5.2 Retour sur `printf`

La fonction `printf` vue précédemment est un `fprintf` sur *standard output*. Les formats admissibles pour `printf` sont donc rigoureusement les mêmes que ceux de `fprintf`.

6.5.3 Exemples d'utilisation des formats

```
            source C                          resultats

printf("|%d|\n",1234);                          |1234|
printf("|%-d|\n",1234);                         |1234|
printf("|%+d|\n",1234);                         |+1234|
printf("|% d|\n",1234);                         | 1234|
printf("|%10d|\n",1234);                       |      1234|
printf("|%10.6d|\n",1234);                     |    001234|
printf("|%10.2d|\n",1234);                     |      1234|
printf("|%.6d|\n",1234);                       |001234|
printf("|%.2d|\n",1234);                       |1234|

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

printf("|%x|\n",0x56ab);                        |56ab|
printf("|%#x|\n",0x56ab);                      |0x56ab|
printf("|%X|\n",0x56ab);                       |56AB|
printf("|%#X|\n",0x56ab);                      |0X56AB|

printf("|%f|\n",1.234567890123456789e5);       |123456.789012|
printf("|%.4f|\n",1.234567890123456789e5);     |123456.7890|
printf("|%.20f|\n",1.234567890123456789e5);    |123456.78901234568000000000|
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);    |1.23456789012345680000e+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|
```

6.5.4 Entrées formattées : `fscanf`

Nous présenterons les fonctionnalités qui sont communes à Unix System V et Unix BSD.

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

`fscanf` (*file-ptr* , *format* , $param_1$, $param_2$, ... , $param_n$)
dans lequel:

1. *file-ptr* est une valeur de type pointeur vers `FILE` qui repère le fichier à partir duquel on veut lire,

2. *format* est une chaîne de caractères,

3. les $param_i$ sont des pointeurs.

`fscanf` lit le fichier défini par *file-ptr* en l'interprétant selon *format*. Le résultat de cette interprétation a pour effet de mettre des valeurs dans les variables pointées par les différents $param_i$.

On définit les termes suivants :

caractères de séparation il s'agit des caractères blanc, horizontal tabulation, et line-feed.

unité (du flot d'entrée): une chaîne de caractères ne contenant pas de caractères de séparation.

La chaîne format contient :

- des caractères de séparation qui, sauf dans les deux cas cités plus loin, font lire à fscanf jusqu'au prochain caractère qui n'est pas un caractère de séparation.
- des caractères ordinaires (autres que %) qui doivent coïncider avec les caractères du flot d'entrée.
- des séquences d'escape introduites par le caractère %. Ces séquences spécifient le travail à effectuer sur le flot d'entrée.

Les séquences d'escape ont la syntaxe suivante :

% $*_{option}$ $nombre_{option}$ *carac*

`*` la présence de ce signe indique que la prochaine unité du flot d'entrée doit être ignorée.

nombre indique la longueur maximum de l'unité à reconnaître.

carac peut prendre l'une des valeurs suivantes :

`d` la prochaine unité sera interprétée comme un nombre décimal.

`o` la prochaine unité sera interprétée comme un nombre octal.

`x` la prochaine unité sera interprétée comme un nombre hexadécimal

Dans les trois cas précédents $param_i$ est interprété comme un pointeur vers un int. De plus, *carac* peut être précédé de la lettre `h` ou `l` pour indiquer que $param_i$ n'est pas un pointeur vers un `int`, mais plutôt un pointeur vers un `short int`, ou un pointeur vers un `long int`.

`c` $param_i$ est interprété comme étant un pointeur vers un caractère. Le prochain caractère du flot d'entrée est mis dans le caractère pointé. Pour ce cas seulement, il n'y a plus de notion de caractère séparateur. On obtient dans le caractère pointé le véritable caractère suivant du flot d'entrée, même s'il s'agit d'un caractère de séparation. Si on désire ignorer les caractères de séparation, il faut utiliser la séquence d'escape `%1c`. Si la séquence d'escape comportait l'indication de longueur maximum de l'unité à reconnaître, c'est ce nombre de caractères qui est lu.

`s` — param$_i$ est interprété comme un pointeur vers une chaîne de caractères. La prochaine unité du flot d'entrée, terminée par un *null* est mise dans la chaîne pointée.

`e,f` — *param$_i$* est interprété comme un pointeur vers un `float`. La prochaine unité du flot d'entrée doit avoir le format d'un nombre flottant (même format qu'une constante flottante du langage C). Ce nombre est affecté au float pointé. Les caractères `e`, ou `f` peuvent être précédés du caractère `l` pour indiquer que *param$_i$* n'est pas un pointeur vers un `float`, mais un pointeur vers un `double`.

`[` — 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 `^` est immédiatement après le crochet ouvrant, le *scanset* est l'ensemble des caractères ne se trouvant pas dans la chaîne entre crochets.

`fscanf` lit dans le fichier repéré par *file-ptr* jusqu'à la rencontre d'un caractère ne faisant pas partie du *scanset*. Comme dans le cas du format `c`, il n'y a plus de notion de caractère séparateur, on ne considère que le *scanset*. *param$_i$* est interprété comme un pointeur vers une suite de caractères dans lesquels on met les caractères lus.

Valeur retournée

En ce qui concerne la valeur retounée, il n'y a malheureusement pas compatibité entre System V et BSD. Dans le cas de fin de fichier, `fscanf` retourne bien `EOF` dans les deux versions d'Unix. Par contre, si `fscanf` n'a pas détecté une fin de fichier, il retourne le nombre d'unités qui ont été affectées dans le cas de System V, et le nombre d'unités illégales dans le cas de Unix BSD. Dans ce dernier cas, il est impossible de connaître le nombre d'unités qui ont étées affectées.

ex:

```
(void)fscanf(fi,"%d %d",&i,&j)
```

si le flot d'entrée contient `12 34`, les valeurs 12 et 34 seront affectées respectivement à `i` et `j`.

6.6 Exercice

Soit un fichier de données structuré en une suite de lignes contenant chacune un nom de personne, un nom de piece, un nombre et un prix. ex:
dupond villebrequin 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 iteration 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$\times$prix.

```
#include "stdio.h"

/*****************************************************************************/
/*                                                                           */
/*                              main                                         */
/*                                                                           */
/*****************************************************************************/
void 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);
   }
}
```

Chapter 7

Structures et unions

7.1 Notion de structure

Il est habituel en programmation que l'on ait 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 de sécurité sociale, etc...

La réponse à ce besoin est la notion de structure.

7.2 Déclaration de structure

Voyons sur un exemple:

```
struct personne
   {
   char nom[20];
   char prenom[20];
   int no_ss;
   };
```

Cette déclaration ne déclare aucune variable. Elle déclare simplement l'identificateur `personne` comme étant le nom d'un type de structure composée de trois champs, 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_ss`.

On pourra par la suite déclarer des variables d'un tel type de la manière suivante:

```
struct personne p1,p2;
```

ceci déclare deux variables de type `struct personne` de noms `p1` et `p2`;

La méthode qui vient d'être exposée est la méthode recommandée, mais on peut utiliser des variantes.

On peut déclarer des variables de type struct sans donner un nom au type de la structure, par exemple:

```
struct
```

```
   {
   char nom[20];
   char prenom[20];
   int no_ss;
   }p1,p2;
```

déclare deux variables de noms `p1` et `p2` comme étant deux structures de trois champs dont le premier est ..., mais elle ne donne pas de nom au type de la structure. Ce qui à pour conséquence que si, à un autre endroit du programme, on désire déclarer une autre variable `p` du même type, il faudra écrire:

```
struct
   {
   char nom[20];
   char prenom[20];
   int no_ss;
   }p;
```

Un autre variante consiste à donner à la fois un nom à la structure, et à déclarer une ou plusieurs variables. Par exemple:

```
struct personne
   {
   char nom[20];
   char prenom[20];
   int no_ss;
   }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;
```

La premiere méthode est recommandée, car elle permet de bien séparer la définition du type structure de ses utilisations.

7.3 Accès aux champs des structures

Pour désigner un champ d'une structure, il faut utiliser l'opérateur de sélection de champ qui se note `.` (point).

Par exemple, si `p1` et `p2` sont deux variables de type `struct personne`, on désignera le champ `nom` de `p1` par `p1.nom`, et on désignera le champ `no_ss` de `p2` par `p2.no_ss`.

Les champs 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]`.

7.4 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 dejà 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`.

7.5 Exercice

Soit un fichier de données identiques à celui de l'exercice précédent.

Ecrire 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                                     */
/*************************************************************************/
void 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 apres 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-dimentionne\n");
   else
      {
      /*   impression des commandes memorisees   */
      /*   -----------------------------------   */
      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);
      }
   }
}
```

7.6 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
   {
   ...
   };

void 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;
}
```

7.7 Structures dont un des champs 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 champ qui soit de type pointeur vers une structure.

Cela se fait de la façon suivante:

```
struct personne
   {
   ...  /*   les differents champs   */
   struct personne *suivant;
   };
```

le champ de nom `suivant` est déclaré comme étant du type pointeur vers une `struct personne`.

7.8 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 champ de la structure pointée par `p`?.

Etant donné que `*p` désigne la structure, on serait tenté d'écrire `*p.nom` pour référencer par exemple le champ `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 n'aurait de sens que si `p` etait une structure dont un des champs s'appelerait `nom` et serait un pointeur).

Dans notre cas, il faut écrire `(*p).nom` pour forcer à ce que l'indirection se fasse avant la sélection.

Etant donné que cette écriture est assez lourde, le langage C a prévu un nouvel opérateur noté `->` qui réalise à la fois l'indirection et la sélection, de façon à ce que `p -> nom` soit identique à `(*p).nom`.

exemple: si `p` est de type pointeur vers la `struct personne` définie précédement, pour affecter une valeur au champ `no_ss` de la structure pointée par `p`, on peut écrire:

```
p -> no_ss = 1345678020;
```

7.9 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* `)`

ex:

```
int i,taille;

taille = sizeof i;
taille = sizeof (short int);
taille = sizeof (struct personne);
```

7.10 Allocation et libération d'espace pour les structures

7.10.1 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:

```
struct personne *p;

p = malloc(sizeof(struct personne));
```

Allocation d'un tableau d'éléments: fonction `calloc`

Elle admet deux paramètres:

1. le premier est le nombre d'éléments désirés.
2. 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:

```
struct personne *p;
int nb_elem;

/*   determination de nb_elem   */
p = calloc(nb_elem,sizeof(struc personne));
```

On peut alors utiliser les éléments `p[0]`, `p[1]`, ... `p[nb_elem-1]`.

7.10.2 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 précedemment rendu par un appel à `malloc` ou `calloc`.

Utilisation typique:

```
struct personne *p;

p = malloc(sizeof(struct personne));
/*   utilisation de la structure allouee   */
free(p);
```

7.11 Exercice

Faire la même chose que dans l'exercice précédent, mais en mémorisant le fichier de données, non pas dans un tableau de structures, mais dans une liste de structures chaînées.

```
#include <stdio.h>

/*************************************************************************/
/*                                                                       */
/*                              main                                     */
/*                                                                       */
/*   Remarque:                                                           */
/*      Le compilateur emet un avertissement au sujet de l'affectation   */
/*      cour = malloc(...) mais le code genere est ok                    */
/*                                                                       */
/*************************************************************************/
void main()
{
FILE * fi;
struct commande
   {
   char nom[80];
   char article[80];
   int nombre,prix;
   struct commande *suiv;
   };

struct commande *l_com = NULL;   /*   liste des commandes   */
struct commande *prec,*cour;   /*   pour la commande precedente et courante   */
int val_ret;                   /*   valeur de retour de fscanf                */

/*   open du fichier   */
/*   ---------------   */
if ((fi = fopen("exer7.data","r")) == NULL)
   printf("Impossible d'ouvrir le fichier exer7.data\n");
else
   {
   /*   lecture du fichier avec creation de la liste chainee   */
   /*   ----------------------------------------------------   */
   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)
         printf("%s %s %d %d\n",
                cour -> nom,cour -> article,cour -> nombre,cour -> prix);
      }

   /*   fermeture du fichier   */
   /*   --------------------   */
   fclose(fi);
   }
}
```

7.12 Passage de structures en paramètre

Le langage C a commme contrainte d'interdire de passer une structure en paramètre à une procédure ou une fonction, mais rien par contre, n'empêche de passer en paramètre un pointeur vers une structure.

Voyons sur un exemple:

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:

```
#define AVANT 1
#define IDEM  2
#define APRES 3

int cmp_date(pd1,pd2)
struct date *pd1,*pd2;
{
if (pd1 -> annee > pd2 -> annee)
   return(APRES);
else if (pd1 -> annee < pd2 -> annee)
   return(AVANT);
else
   {
   /*   comparaison portant sur mois et jour   */
   }
}
```

Une utilisation de cette fonction pourra être:

```
struct date d1,d2;

if (cmp_date(&d1,&d2) == AVANT)
   ...
```

7.13 Structures retournées par une fonction

Il existe la même contrainte concernant les valeurs retournées par une fonction que pour les paramètres, à savoir qu'une fonction ne peut pas retourner une structure, mais peut retourner un pointeur vers une structure.

Avec l'exemple des `struct date`, on peut définir une fonction retournant un pointeur repérant une `struct date` initialisée au premier janvier de l'an 2000 de la façon suivante:

```
struct date *f()   /* f fonction retournant un pointeur vers une struct date */
{
```

```
struct date *p;

p = malloc(sizeof(struct date));
p -> jour = 1; p -> mois = 1; p-> annee = 2000;
return(p);
}
```

7.14 Déclaration de procédures / fonctions externes

Lorsque dans les exemples précédents, on a écrit des choses du genre:

```
struct date *p;

p = malloc(sizeof(struct date));
```

on s'est apercu, lors de la compilation, qu'il y avait émission d'un avertissement concernant l'instruction `p = malloc(...)`.

Cet avertissement précisait que le type de la valeur affectée n'était pas compatible avec celui de la variable à laquelle on l'affectait.

Le code généré était parfaitement correct, mais un tel message est toujours inquiétant, et il vaut toujours mieux programmer de manière à le faire disparaitre.

Pour comprendre la raison de ce message il faut savoir 2 choses:

1. Les procédures et fonctions `printf`, `scanf`, `malloc`, `free` etc... ne sont pas en C, contrairement à ce qu'elles sont dans la majorité des autres langages, des procédures connues du langage. Ce sont des procédures et fonctions externes, indiscernables du point de vue du langage, des autres procédures et fonctions écrites par l'utilisateur.

2. Lorsqu'on rencontre dans un source C une occurence d'une utilisation d'une fonction avant sa définition, le langage considère par défaut que cette fonction retourne un `int`. Si tel est bien le cas, il n'y a pas de problème, sinon, il sera nécessaire de déclarer cette fonction de manière à informer le compilateur du type des valeurs retournées par la fonction.

   Dans notre exemple, dans la partie des déclarations externes à toutes procédures et fonctions, on écrirait:

   ```
   extern struct date *malloc();
   ```

   ce qui déclarerait la fonction `malloc` comme retournant un pointeur vers une `struct date` et ne provoquerait plus d'émission de message d'avertissement lorsqu'on affecterait ultérieurement le résultat de `malloc()` à une variable de type pointeur vers une `struct date`.

7.15 Exercice

Modifier le programme précédent:

1. en écrivant une procédure d'impression d'une `struct commande`, procédure qui admettra en paramètre un pointeur vers une telle structure.

2. en écrivant une fonction de recherche de commande maximum (celle pour laquelle le produit nombre $\times$ 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 <stdio.h>

/*   les types communs a toutes les procedures   */
/*   ----------------------------------------   */
struct commande
   {
   char nom[80];
   char article[80];
   int nombre,prix;
   struct commande *suiv;
   };

/*   les procedures / fonctions externes   */
/*   ----------------------------------   */
extern struct commande *malloc();

/*************************************************************************/
/*                                                                       */
/*                              print_com                                */
/*                                                                       */
/*   But:                                                                */
/*      Imprime une structure commande                                   */
/*                                                                       */
/*************************************************************************/
void print_com(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 recherchee       */
/*                      ou NULL si l_com est vide                            */
/*                                                                           */
/*****************************************************************************/
struct commande *max_com(l_com)
struct commande *l_com;

{
struct commande *pmax;   /*   pointeur vers le max courant   */
struct commande *cour;   /*   pointeur vers l'element courant   */
int vmax,vcour;

if (l_com == NULL)
   return(NULL);
else
   {
   pmax = l_com; vmax = (pmax -> nombre) * (pmax -> prix);

   for (cour = l_com -> suiv; cour != NULL; cour = cour ->suiv)
      {
      vcour = (cour -> nombre * cour -> prix);
      if (vcour > vmax)
         {
         vmax = vcour;
         pmax = cour;
         }
      }

   return(pmax);
   }
}
```

```
/*****************************************************************************/
/*                                                                           */
/*                              main                                         */
/*                                                                           */
/*****************************************************************************/
void main()
{
FILE * fi;
struct commande *l_com = NULL; /*   liste des commandes                     */
struct commande *prec,*cour;   /*   pour la commande precedente 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 creation 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));
      }
```

```
   /*   fermeture du fichier   */
   /*   --------------------   */
   fclose(fi);
   }
}
```

7.16 Les champs de bits

7.16.1 Généralités

Il est parfois nécessaire pour le programmeur de décrire la structure d'un mot machine, cela pour plusieurs raisons:

- un mot de l'espace mémoire est un registre de coupleur découpé en différents champs de bits.
- pour des raisons de gain de place, on désire faire cœxister plusieurs variables à l'intérieur d'un entier.

Il existe dans le langage C un moyen de réaliser cela, à l'aide du concept de structure. En effet, dans une déclaration de structure, il est possible de faire suivre la définition d'un champ par une indication du nombre de bits que doit avoir ce champ.

Voyons sur un exemple:

```
struct etat
   {
   unsigned int sexe : 1;
   unsigned int situation : 3;
   unsigned int invalide : 1;
   };
```

La `struct etat` voit son premier champ implémenté sur 1 bit, son second sur 3 bits et le dernier sur 1 bit.

7.16.2 Contraintes

1. un champ de bits ne peut pas être d'une taille supérieure à celle d'un `int`.
2. un champ de bits ne peut pas être à cheval sur deux `int`.
3. l'ordre dans lequel sont mis les champs de bits à l'intérieur d'un mot dépend du compilateur. Si on utilise les champs de bits pour gagner de la place, cela n'est pas génant. Dans le cas ou on les utilise pour décrire une ressource matérielle de la machine (registre de coupleur, mot d'etat programme etc..) il est bien sûr nécessaire de connaître le comportement du compilateur.
4. un champ de bit déclaré comme étant de type `int`, peut en fait se comporter comme un `int` ou comme un `unsigned int` (cela dépend du compilateur). Il est donc recommandé d'une manière générale de déclarer les champs de bits comme étant de type `unsigned int`.
5. un champ de bits n'a pas d'adresse, on ne peut donc pas lui appliquer l'opérateur adresse de (`&`).

7.17 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 réels dès que ce ne sera plus le cas. Il sera nécessaire de disposer de variables pouvant prendre soit des valeurs entières, soit des valeurs réelles.

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 simplememt remplacé par le mot clé `union`.

ex:

```
union nombre
   {
   int i;
   float f;
   }
```

La différence sémantique entre les `struct` et les `unions` est la suivante: alors que pour une variable de type structure tous les champs peuvent avoir en même temps une valeur, une variable de type union ne peut avoir, à un instant donné, qu'un seul champ ayant une valeur.

Dans l'exemple précédent, si on déclare la variable `n` comme étant de type union nombre par:

```
union nombre n;
```

cette variable pourra posséder soit une valeur entière, soit une valeur réelle, mais pas les deux à la fois.

7.18 Accès aux champs de l'union

Cet accès se fait avec le même opérateur sélection (noté `.`) que celui qui sert à accéder aux champs des structures.

Dans l'exemple précédent, si on désire faire posséder à la variable `n` une valeur entière, on pourra écrire:

```
n.i = 10;
```

si on désire lui faire posséder une valeur réelle, on pourra écrire:

```
n.f = 3.14159;
```

7.19 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 champ 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 champ 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 procèderait de la manière suivante:

```
#define ENTIER 0
#define REEL 1

struct arith
   {
   int typ_val;   /*   peut prendre les valeurs ENTIER ou REEL   */
   union
      {
      int i;
      float f;
      } u;
   };
```

la `struct arith` a deux champs `typ_val` de type `int`, et `u` de type union d'`int` et de `float`.

on déclarerait des variables par:

```
struct arith a1,a2;

puis on pourrait les utiliser de la maniere suivante:

a1.typ_val = ENTIER;
a1.u.i = 10;

a2.typ_val = REEL;
a2.u.f = 3.14159;
```

Si on passait en paramètre à une procédure un pointeur vers une `struct arith`, la procédure testerait la valeur du champ `typ_val` pour savoir si l'union reçue possède un entier ou un réel.

7.20 Une méthode pour alléger l'accès aux champs

Quand une union est dans une structure, il faut donner un nom au champ de la structure qui est de type union, ce qui à pour conséquence de rendre assez lourd l'accès aux champs de l'union.

Dans l'exemple précédent, il faut écrire `a1.u.f` pour accéder au champ `f`.

On peut alléger l'écriture en utilisant les facilités du pré-processeur. 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;
```

Chapter 8

Les expressions

Avant de passer à l'étude de la sémantique des expressions, nous allons présenter les quelques opérateurs non encore vus jusqu'à présent.

8.1 Les opérateurs

8.1.1 Opérateur *non bit à bit*

- Syntaxe:

  expression :
  ⇒ `~` *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 `~`.

8.1.2 Opérateur *et bit à bit*

- Syntaxe:

  expression :
  ⇒ *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 `&`.

8.1.3 Opérateur *ou bit à bit*

- Syntaxe:

  expression :
  ⇒ *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 |.

8.1.4 Opérateur *décalage à gauche*

- Syntaxe:

  expression :
  ⇒ $expression_1$ << $expression_2$

- 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 <<.

8.1.5 Opérateur *décalage à droite*

- Syntaxe:

  expression :
  ⇒ $expression_1$ >> $expression_2$

- 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.

8.1.6 Opérateur conditionnel

- Syntaxe:

  expression :
  ⇒ $expression_1$? $expression_2$: $expression_3$

- 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.

8.1.7 Opérateur *virgule*

- Syntaxe:

  expression :
  ⇒ $expression_1$, $expression_2$

- Sémantique:

  $expression_1$ est évaluée et sa valeur ignorée. $expression_2$ est évaluée et sa valeur est la valeur de l'expression virgule.

Remarque

Etant 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.

8.1.8 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 :

  $\Rightarrow$ *lvalue* `+=` *expression*

- Sémantique:

  lvalue `=` *lvalue* `+` *expression*

8.2 Opérateur *forceur*

- Syntaxe:

  expression :

  $\Rightarrow$ `(` *type* `)` *expression*

- Sémantique: *expression* est évaluée et convertie dans le type indiqué par *type*.

Exemples d'utilisation

1. conversion lors d'un passage de paramètre.

   Si une procédure `p` a un paramètre de type `long int`, et si on désire lui passer la valeur possédée par la variable `i` de type `int`, il ne faut pas écrire `p(i)` mais `p((long int) i)`.

2. utilisation de `malloc`

   Supposons que nous ayons plusieurs types de structures a allouer par `malloc`. Nous pouvons définir `malloc` de la manière suivante:

   ```
   extern char * malloc();
   ```

   et lors de l'appel de `malloc` utiliser un forceur pour qu'il n'y ait pas d'émission d'avertissement.

   ```
   p1 = (struct s1 *) malloc(sizeof(struct s1));
   p2 = (struct s2 *) malloc(sizeof(struct s2));
   ```

8.3 Sémantique des expressions

8.3.1 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 connait les priorités relatives des opérateurs `*` et `++`.

8.3.2 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	`()` `[]` `->` `.`	
15	`++` [a] `--` [b]	
14	`!` `~` `++` [c] `--` [d] `-` [e] *forceur* `*` [f] `&` [g] `sizeof`	
13	`*` [h] `/` `%`	G
12	`+` `-`	G
11	`<<` `>>`	G
10	`<` `<=` `>` `>=`	G
9	`==` `!=`	G
8	`&` [i]	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] indirection
[g] adresse de
[h] multiplication
[i] et bit bit

Attention

Le choix fait pour les priorités des opérateurs concernant les opérations bits à bits est assez désastreux, en ce sens que `a & b == c` signifie `a & (b == c)` ce qui est rarement ce que veut le programmeur ! Attention donc à parenthéser convenablement les sous-expressions.

8.3.3 Ordre d'évaluation des opérandes

A 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`.

Chapter 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 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'a 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 apparait.

ex:

```
int i;          /*   declaration a l'exterieur de toute fonction          */

void proc1(j)   /*   debut de proc1                                       */
int j;          /*   parametre de la procedure   proc1                    */
{

...             /*   instructions 1                                       */

k:
if (...)
   {
   int l;       /*  declaration a l'interieur d'une instruction composee  */

   ...          /*   instructions 2                                       */
   }
```

```
}               /*   fin de proc1                                    */

int func1()     /*   debut 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 $instructions_2$.

9.2 Visibilité des identificateurs

Dans le langage C, l'emboîtement 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 surcroit, les déclarations d'une instruction composée cachent les déclarations de même nom, qui sont à l'extérieur de toute fonction.

ex:

```
int i;
int j;

void proc1()
{
int i;      /*   cache le i precedent   */
int k;

if (a > b)
   {
   int i;   /*   cache le i precedent   */
   int j;   /*   cache le j precedent   */
   int k;   /*   cache le k precedent   */

   ...
   }
}
```

9.3 Les espaces de noms

9.3.1 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;
   };

void main()
{
struct st1 s1;   /*   declaration de la variable s1   */
struct st2 s2;   /*   declaration 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.

9.3.2 Les espaces de noms du langage C

Il y a trois espaces de noms dans le langage:

- Les étiquettes de structures ou unions. Ce sont les identificateurs qui suivent immédiatement les mots-clés `struct` et `union`.

  ex:

  ```
  struct date
     {
     int jour,mois,annee;
     } d1,d2;
  ```

  ici, `date` est une étiquette de structure, `d1` et `d2` étant des noms de variable de type `struct date`.

- Les noms de champs de structures ou unions. Il y a un espace de nom pour chaque structure et chaque union. Ceci permet donc d'avoir des noms de champs identiques dans des structures ou unions différentes. Un nom de champ est toujours suivi soit de l'opérateur "sélection" (`.`), soit de l'opérateur "indirection et sélection" (`->`).

- 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 4 espaces de noms différents.

```
int i;          /*   i est un nom didentificateur       */

struct i        /*   i est une etiquette 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;

void main()
{
i = 1;
i1.i = 2;
ii1.i = 3;
}
```

Remarques

1. Certains compilateurs considèrent qu'il existe 4 classes de noms en faisant une classe particulière pour les noms d'étiquettes d'instructions. Ceci est la stratégie adoptée par le compilateur UNIX SystemV. Cependant, si on s'en tient à trois espaces de noms, le programmeur est sûr d'écrire des programmes portables entre UNIX System V et UNIX Berkeley. C'est la raison pour laquelle nous avons adopté cette vision des choses.
2. Certains auteurs considèrent egalement 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.

9.4 Durée de vie

Du point de vue de la durée de vie, il existe deux types de variables: les variables *statiques* et le variables *automatiques*.

- 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 dans une instruction composée, et libérées lors de la sortie de l'instruction composée.

On pourrait considérer qu'il existe un troisième type de variables, les variables *dynamiques*, dont la création et la destruction sont explicitement réalisées par le programmeur à l'aide des fonctions `malloc` et `free`. Nous n'avons cependant pas retenu cette vision des choses, dans la mesure où ces deux fonctions font partie de la librairie standard et non pas du langage *stricto sensu*.

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 suivant.

Imaginons une procédure qui pour une raison quelconque doit connaître combien de fois elle a étée 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:

```
ex:
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.

9.5 Classes de mémoire

9.5.1 Position du problème

Lors de l'exécution d'un programme écrit en C il y a trois zones de mémoire différentes:

- 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 crée 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).

9.5.2 Les indicateurs de classe de mémoire

Il existe 4 mots-clés du langage que la grammaire nomme *indicateurs de classe de mémoire.* Il s'agit des mots-clés suivants:

auto Cet indicateur 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.

ex:

```
{
auto int i;
...
}
```

static Cet indicateur 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 .

ex:

```
static int i;      /*   i ne sera pas exporte par l'editeur de liens   */
int j;             /*   j sera exporte par l'editeur de liens          */

static void f()    /*   f ne sera pas exporte par l'editeur de liens   */
{
static int k;      /*   k aura une duree de vie globale                */
...
}

void g()           /*   g sera exportee par l'editeur de liens         */
{
...
}
```

register Cet indicateur 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 la latitude pour le compilateur d'allouer pour la variable une ressource à accès rapide.

extern Cet indicateur 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.

9.5.3 Indicateur de classe mémoire par défaut

La grammaire autorise l'omission de l'indicateur de classe mémoire lorsqu'on fait une déclaration. Un indicateur est alors pris par défaut selon les règles suivantes:

1. `extern` est pris par défaut pour toutes les déclarations de fonctions et toutes les déclarations de variables externes à toute fonction.
2. `auto` est pris par défaut pour les déclarations de variables locales à une instruction composée.
3. `extern` est pris par défaut pour toute déclaration de fonction interne à une instruction composée.

Discussion

Que le lecteur ne s'inquiète pas si il a du mal a avoir en première lecture des idées nettes sur cette notion de *indicateurs de classe mémoire*, car ce n'est pas une chose très heureuse dans le langage. 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 là il n'y a rien à dire, par contre, 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, c'est à dire 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. `extern` n'a rien à voir avec un quelconque problème de classe mémoire, il sert à **importer** le nom d'une variable.

9.6 Syntaxe des déclarations

La grammaire des déclarations est la suivante:
declaration :

⇒ *indicateur-de-classe*$_{option}$ *indicateur-de-type* *liste-de-déclareur-init* ;

indicateur-de-classe :

⇒ `auto`
⇒ `extern`
⇒ `static`
⇒ `register`

indicateur-de-type :
⇒ `char`
⇒ `unsigned char`
⇒ `short int`
⇒ `int`
⇒ `long int`
⇒ `unsigned short int`
⇒ `unsigned int`
⇒ `unsigned long int`
⇒ *indicateur-de-struct-ou-union*

indicateur-de-struct-ou-union :
⇒ `struct` `{` *liste-de-struct-decl* `}`
⇒ `struct` *identificateur* `{` *liste-de-struct-decl* `}`
⇒ `struct` *identificateur*
⇒ `union` `{` *liste-de-struct-decl* `}`
⇒ `union` *identificateur* `{` *liste-de-struct-decl* `}`
⇒ `union` *identificateur*

liste-de-struct-decl :
⇒ *decl-de-struct*
⇒ *decl-de-struct* *liste-de-struct-decl*

decl-de-struct :
⇒ *indicateur-de-type* *liste-de-déclareur-de-struct*

liste-de-déclareur-de-struct :
⇒ *déclareur-de-struct*
⇒ *déclareur-de-struct* `,` *liste-de-déclareur-de-struct* `;`

déclareur-de-struct :
⇒ *déclareur*
⇒ *déclareur* `:` *expression-constante*
⇒ `:` *expression-constante*

liste-de-déclareur-init :
⇒ *déclareur-init*
⇒ *déclareur-init* `,` *liste-de-déclareur-init*

déclareur-init :
⇒ *déclareur* $initialiseur_{option}$

déclareur :
⇒ *identificateur*
⇒ `*` *déclareur*
⇒ *déclareur* `(` `)`
⇒ *déclareur* `[` *expression-constante*$_{option}$ `]`
⇒ `(` *déclareur* `)`

initialiseur :
⇒ `=` *expression*
⇒ `=` `{` *liste-d'initialiseur* `}`
⇒ `=` `{` *liste-d'initialiseur* `,` `}`

liste-d'initialiseur :
⇒ *expression*
⇒ *liste-d'initialiseur* `,` *liste-d'initialiseur*
⇒ `{` *liste-d'initialiseur* `}`

Exemples

Dans la déclaration

```
int i,j = 2;
```

`int` est un *indicateur-de-type* et `i,j = 2` est une *liste-de-déclareur-init* composée de deux *déclareur-init*: `i` et `j = 2`. `i` est un *déclareur-init* sans la partie *initialiseur*, donc réduit à un *déclareur* lui-même réduit à un *identificateur*. `j = 2` est un *déclareur-init* comportant un *initialiseur* (`= 2`) et un *déclareur* réduit à un *identificateur* (`j`).

Dans la déclaration

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

`t[10]` est un déclareur formé d'un *déclareur* (`t`), suivi de `[` suivi de l'*expression constante* 10, suivi de `]`.

9.7 Sémantique des déclarations

La partie qui nécessite d'être explicitée est la partie de la grammaire concernant les *déclareur*. C'est la partie suivante:
déclareur :
identificateur
`*` *déclareur*
déclareur `(` `)`
déclareur `[` *expression-constante*$_{option}$ `]`
`(` *déclareur* `)`
La sémantique est la suivante:

- la seule règle de la grammaire qui dérive vers un terminal est la règle:
  déclareur :

identificateur
ce qui fait qu'a l'intérieur de tout *déclareur* se trouve un identificateur. Cet identificateur est le nom de l'objet déclaré par la *déclaration.*

ex:

```
char c;    /*   declaration 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 ...".
     ex:

     ```
     /*   declaration de p de type pointeur vers short int   */
     short int *p;
     ```

  2. `( )` est un constructeur permettant permettant de construire des types "fonction retournant ...".
     ex:

     ```
     /*  declaration de sin de type fonction retournant un double  */
     double sin();
     ```

  3. `[` $expression_{option}$ `]` est un constructeur permettant de construire des types "tableau de ...".
     ex:

     ```
     /*   declaration 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 supèrieure 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:
  déclareur :
  `(` *déclareur* `)`
  permet de parenthéser des *déclareur* 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    */
```

9.8 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 *déclareur*.

ex:

```
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 *déclareur* alors que les deux derniers sont des *indicateur-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())[])()
```

9.9 En pratique

Lorsqu'il s'agit d'écrire ou de comprendre un type compliqué, il est recommandé non pas d'utilier la grammaire des *déclareur*, 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 appeller la fonction délivrée

[1] `f` est une fonction retournant un pointeur vers un tableau de pointeurs vers une fonction retournant un `char`

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`.

[2] A l'utilisation il faudra mettre les paramètres effectifs

Chapter 10

Pour finir

10.1 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.

ex:

```
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 dont le premier champ est ...

ces noms de type sont ensuite utilisables dans les déclarations de variables, exactement comme un type de base.

ex:

```
tab t1,t2;           /*   t1 et t2 tableaux de 10 entiers      */
personne *p1,*p2;    /*   p1 et p2 pointeurs vers des struct   */
```

10.2 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.

10.3 Redéfinition d'un type de base

Il est parfois nécessaire de manipuler des variables qui ne peuvent prendre comme valeurs qu'un sous-ensemble de l'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`.

10.4 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   */
```

10.5 Généralités sur la compilation séparée

Dès que l'on programme une application un peu conséquente, il devient nécessaire pour des raisons pratiques de fractionner le source en plusieurs fichiers. Chaque fichier est compilé séparément, puis les binaires obtenus sont liés à l'aide d'un éditeur de liens pour créer le programme désiré.

Il est nécessaire cependant que le langage offre au programmeur les services suivants:

- possibilité de déclarer des objets (variables et procédures) et:
  1. soit les exporter (les faire connaitre aux autre sources)
  2. soit les cacher (empêcher les autres sources d'y accéder).
- référencer à partir d'un source des objets définis dans d'autres sources.

10.6 La compilation séparée dans le langage C

Nous allons passer en revue comment réaliser les besoins du programmeur.

10.6.1 déclarer des objets et les exporter

Toute déclaration de variable se trouvant à l'extérieur de toute procédure ou fonction, exporte la variable déclarée. De la même façon, toute définition de procédure ou fonction est par défaut exportée.

10.6.2 déclarer des objets et les cacher

Pour cacher une déclaration, il faut la faire précéder du mot-clé `static`, ceci aussi bien pour les variables que pour les procédures et fonctions.

ex:

```
static int i;
static t[10];

static f(i,j)
int a,b;
{
...
};
```

10.6.3 référencer un objet déclaré ailleurs

Il faut pour cela, faire précéder la déclaration du mot clé `extern`, ceci aussi bien pour les variables que pour les procédures et fonctions.

ex:

```
extern int i;
extern t[];   /*   reference a un tableau defini ailleurs : pas de taille   */
extern int f(); /* ref. a une fonction definie ailleurs: pas de parametre */
```

10.7 Le préprocesseur

Toute compilation d'un source C commence par une phase dite de pré-processing au cours de laquelle le compilateur mémorise et exécute un certain nombre de commandes permettant de faire de la substitution d'identificateur, de la macro-substitution, de l'inclusion de source et de la compilation conditionnelle.

10.8 substitution d'identificateur

Lorsque le compilateur rencontre une ligne de la forme

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

il mémorise cette définition, et substitue ensuite toute occurence de *identificateur* par *reste-de-la-ligne.*

10.9 Macro-substitution

Lorsque le compilateur rencontre une ligne de la forme

`#define` *identificateur* `(` *liste-d'identificateurs* `)` `reste-de-la-ligne`

où il n'y a pas de blanc entre *identificateur* **et la parenthèse ouvrante**, il mémorise cette définition et remplace toute occurence de *identificateur* `(` *liste-de-paramètres-effectifs* `)` par *reste-de-la-ligne* dans laquelle les paramètres formels auront étés remplacés par les paramètres effectifs.

ex:

```
#define verif(a,b) if ((a) > (b)) erreur(10)

f()
{
...
verif(i+1,j-1);
...
}
verif(i+1,j-1);
```

sera compilé comme si on avait écrit:

```
if ((i+1) > (j-1)) erreur(10);
```

10.10 Compilation conditionnelle

Lorsque le compilateur rencontre:

#if *expression*
suite-de-lignes
#endif

où *expression* a une valeur connue à la compilation, le compilateur compilera *suite-de-lignes* si cette valeur est non nulle. Si *expression* a une valeur nulle, le compilateur ignorera *suite-de-lignes*.

Lorsque le compilateur rencontre:

#if *expression*
suite-de-lignes$_1$
#else
suite-de-lignes$_2$
#endif

où *expression* a une valeur connue à la compilation, le compilateur compilera *suite-de-lignes*$_1$ et ignorera *suite-de-lignes*$_2$ si cette valeur est non nulle. Si *expression* a une valeur nulle, le compilateur ignorera *suite-de-lignes*$_1$ et compilera *suite-de-lignes*$_2$.

Contents