Claude Delannoy

Programmer en langage C

5e édition

Cours et exercices corrigés

 ISBN : 978-2-212-14010-1

EYROLLES

Table des matières

Avant-propos

Le langage C a été créé en 1972 par Denis Ritchie avec un objectif relativement limité : écrire un système d'exploitation (UNIX). Mais ses qualités opérationnelles l'ont très vite fait adopter par une large communauté de programmeurs.

Une première définition de ce langage est apparue en 1978 avec l'ouvrage de Kernighan et Ritchie : *The C programming language*. Mais ce langage a continué d'évoluer après cette date à travers les différents compilateurs qui ont vu le jour. Son succès international a contribué à sa normalisation, d'abord par l'ANSI (American National Standard Institute), puis par l'ISO (International Standards Organization), plus récemment en 1993 par le CEN (Comité européen de normalisation) et enfin, en 1994, par l'AFNOR. En fait, et fort heureusement, toutes ces normes sont identiques, et l'usage veut qu'on parle de « C ANSI » ou de « C norme ANSI ».

La norme ANSI élargit, sans la contredire, la première définition de Kernighan et Ritchie. Outre la spécification de la syntaxe du langage, elle a le mérite de fournir la description d'un ensemble de fonctions qu'on doit trouver associées à tout compilateur C sous forme d'une bibliothèque standard. En revanche, compte tenu de son arrivée tardive, cette norme a cherché à « préserver l'existant », en acceptant systématiquement les anciens programmes. Elle n'a donc pas pu supprimer certaines formulations quelque peu désuètes ou redondantes. Par exemple, la première définition de Kernighan et Ritchie prévoit qu'on déclare une fonction en précisant uniquement le type de son résultat. La norme autorise qu'on la déclare sous forme d'un « prototype » (qui précise en plus le type de ses arguments) mais ne l'impose pas. Notez toutefois que le prototype deviendra obligatoire en C++.

Cet ouvrage a été conçu comme un cours de programmation en langage C. Suivant notre démarche habituelle, héritée de notre expérience de l'enseignement, nous présentons toujours les notions fondamentales sur un ou plusieurs exemples avant d'en donner plus formellement la portée générale. Souvent constitués de programmes complets, ces exemples permettent l'auto-expérimentation.

La plupart des chapitres de cet ouvrage proposent des exercices que nous vous conseillons de résoudre d'abord sur papier, en comparant vos solutions avec celles fournies en fin de volume et en réfléchissant sur les différences de rédaction qui ne manqueront pas d'apparaître. Ils serviront à la fois à contrôler les connaissances acquises et à les appliquer à des situations variées.

Nous avons cherché à privilégier tout particulièrement la clarté et la progressivité de l'exposé. Dans cet esprit, nous avons systématiquement évité les « références avant », ce qui, le cas échéant, autorise une étude séquentielle ; de même, les points les plus techniques ne sont exposés qu'une fois les bases du langage bien établies (une présentation prématurée serait perçue comme un bruit de fond masquant le fondamental).

D'une manière générale, notre fil conducteur est ce qu'on pourrait appeler le « C moderne », c'est-à-dire non pas la norme ANSI pure et dure, mais plutôt l'esprit de la norme dans ce

qu'elle a de positif. Nous pensons ainsi forger chez le lecteur de bonnes habitudes de programmation en C et, par la même occasion, nous lui facilitons son entrée future dans le monde du C++.

Enfin, outre son caractère didactique, nous avons doté cet ouvrage d'une organisation appropriée à une recherche rapide d'information :

- ses chapitres sont fortement structurés : la table des matières, fort détaillée, offre de nombreux points d'entrée,
- au fil du texte, des encadrés viennent récapituler la syntaxe des différentes instructions,
- une annexe fournit la description des fonctions les plus usitées de la bibliothèque standard (il s'agit souvent d'une reprise d'informations déjà présentées dans le texte),
- un index détaillé permet une recherche sur un point précis ; il comporte également, associé à chaque nom de fonction standard, le nom du fichier en-tête (.h) correspondant.

Remarque concernant cette nouvelle édition :

L'ISO a publié deux extensions de la norme : en 1999 (référence ISO/IEC 9899:1999) et en 2011 (référence ISO/IEC 9899.201x). Elles sont connues sous les sigles C99 et C11. Celles-ci sont loin d'être implémentées dans leur totalité par tous les compilateurs.

Dans cette nouvelle édition :

- la mention C ANSI continue à désigner l'ancienne norme, souvent baptisée C89 ou C90 ;
- lorsque cela s'est avéré justifié, nous avons précisé les nouveautés introduites par la norme C99 ou par la norme C11.

À propos de la fonction main

En théorie, selon la norme ANSI, la fonction `main` qui, contrairement aux autres fonctions, ne dispose pas de prototype, devrait posséder l'un des deux en-têtes suivants :

```
int main (void)
int main (int arg, char *argv[])
```

En fait, tant que l'on ne cherche pas à utiliser les « arguments de la ligne de commande », les deux formes :

```
int main()
```

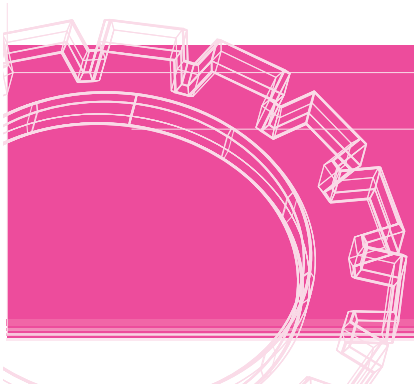
```
main()
```

sont accceptées par toutes les implémentations, la seconde s'accompagnant toutefois fréquemment d'un message d'avertissement.

La première est la plus répandue et c'est celle que nous utiliserons généralement, par souci de simplicité.

Chapitre 1

Généralités sur le langage C



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

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

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

1.1 Un exemple de programme en langage C

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

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

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

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

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

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

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

1.2 Structure d'un programme en langage C

La ligne :

```
int main()
```

se nomme un « en-tête ». Elle précise que ce qui sera décrit à sa suite est en fait le *programme principal* (`main`). Lorsque nous aborderons l'écriture des fonctions en C, nous verrons que celles-ci possèdent également un tel en-tête ; ainsi, en C, le programme principal apparaîtra en fait comme une fonction dont le nom (`main`) est imposé. Notez qu'ici nous utilisons un en-tête simplifié. Les différents en-têtes possibles de la fonction `main` seront présentés aux paragraphes 13 du chapitre 16 et 11.2 du chapitre 8.

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

1.3 Déclarations

Les trois instructions :

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

sont des « déclarations ».

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

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

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

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

1.4 Pour écrire des informations : la fonction printf

L'instruction :

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

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

```
"Bonjour\n"
```

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

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

L'instruction suivante :

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

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

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

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

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

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

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

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

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

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

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

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

- avant de commencer cette répétition, réaliser :

```
i = 0
```

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

```
i < NFOIS
```

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

```
 i++
```

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

```
i = i + 1
```

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

1.6 Pour lire des informations : la fonction `scanf`

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

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

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

```
"%f"
```

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

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

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

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

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

Les lignes :

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

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

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

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

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

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

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

Remarque

Une instruction telle que :

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

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

Notez que C dispose de trois sortes d'instructions :

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

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

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

1.8 Les directives à destination du préprocesseur

Les trois premières lignes de notre programme :

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

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

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

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

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

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

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


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

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

Remarque

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

1.9 Un second exemple de programme

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

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

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

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

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

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

Remarques

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

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

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

Au lieu de :

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

on pourrait écrire :

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

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

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

```
putchar (op) ;
```

affiche le caractère contenu dans `op`.

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

2 Quelques règles d'écriture

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

2.1 Les identificateurs

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

En ce qui concerne les lettres :

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

```
lg_lig    valeur_5    _total    _89
```

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

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

2.2 Les mots-clés

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

Les mots-clés du langage C

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

2.3 Les séparateurs

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

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

Ainsi, vous devrez impérativement écrire :

```
int x,y
```

et non :

```
intx,y
```

En revanche, vous pourrez écrire indifféremment :

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

ou plus lisiblement :

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

2.4 Le format libre

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

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

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

Exemple de programme mal présenté

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

2.5 Les commentaires

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

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

Voici quelques exemples de commentaires :

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

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

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

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

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

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

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

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

3 Création d'un programme en langage C

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

3.1 L'édition du programme

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

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

3.2 La compilation

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

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

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

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

3.3 L'édition de liens

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

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

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

3.4 Les fichiers en-tête

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

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

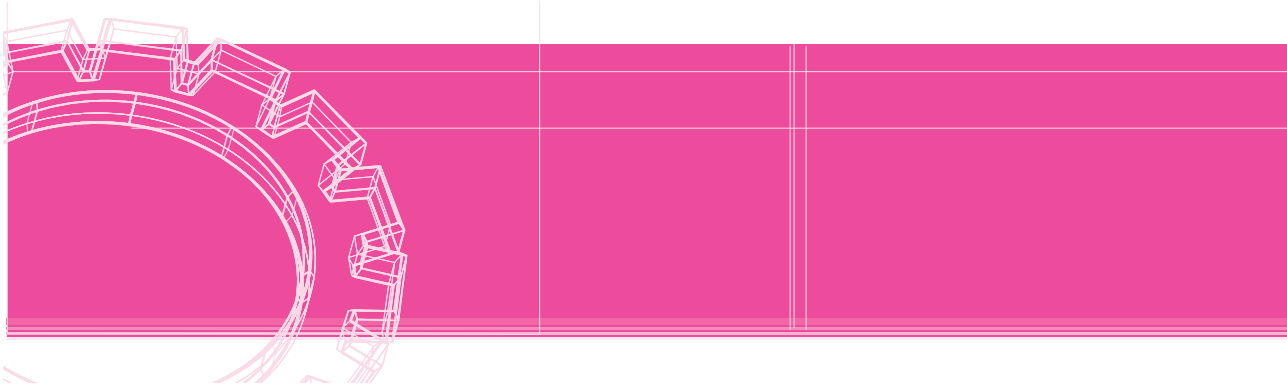
Lorsqu'on écrit un programme, on ne fait pas toujours la différence entre fonction et macro, puisque celles-ci s'utilisent de la même manière. Toutefois, les fonctions et les macros sont traitées de façon totalement différente par l'ensemble « préprocesseur + compilateur + éditeur de liens ».

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

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

Chapitre 6

La programmation modulaire et les fonctions



Comme tous les langages, C permet de découper un programme en plusieurs parties nommées souvent « modules ». Cette programmation dite modulaire se justifie pour de multiples raisons :

- Un programme écrit d'un seul tenant devient difficile à comprendre dès qu'il dépasse une ou deux pages de texte. Une écriture modulaire permet de le scinder en plusieurs parties et de regrouper dans le programme principal les instructions en décrivant les enchaînements. Chacune de ces parties peut d'ailleurs, si nécessaire, être décomposée à son tour en modules les plus élémentaires ; ce processus de décomposition pouvant être répété autant de fois que nécessaire, comme le préconisent les méthodes de programmation structurée.
- La programmation modulaire permet d'éviter des séquences d'instructions répétitives, et cela d'autant plus que la notion d'argument permet de paramétrer certains modules.
- La programmation modulaire permet le partage d'outils communs qu'il suffit d'avoir écrits et mis au point une seule fois. Cet aspect sera d'autant plus marqué que C autorise effectivement la compilation séparée de tels modules.

1 La fonction : la seule sorte de module existant en C

Dans les anciens langages, on trouve deux sortes de modules, à savoir :

- Les **fonctions**, assez proches de la notion mathématique correspondante. Notamment, une fonction dispose d'arguments corrrespondant à des informations qui lui sont transmises et elle fournit un résultat unique. Désigné par le nom même de la fonction, ce résultat peut alors apparaître dans une expression. On dit d'ailleurs que la fonction possède une valeur et qu'un appel de fonction est assimilable à une expression.
- Les **procédures** (terme Pascal) ou **sous-programmes** (terme Fortran 90 ou Visual Basic) qui élargissent la notion de fonction. La procédure ne possède plus de valeur à proprement parler et son appel ne peut plus apparaître au sein d'une expression. Par contre, elle dispose toujours d'arguments. Parmi ces derniers, certains peuvent, comme pour la fonction, correspondre à des informations qui lui sont transmises. Mais d'autres, contrairement à ce qui se passe pour la fonction, peuvent correspondre à des informations qu'elle produit en retour de son appel. De plus, une procédure peut réaliser une action, par exemple afficher un message (en fait, dans la plupart des langages, la fonction peut quand même réaliser une action, bien que ce ne soit pas là son rôle).

En C, il n'existe qu'une seule sorte de module, nommé **fonction**. Il en ira de même, non seulement en C++, C# ou Java (langages dont la syntaxe est très proche de celle de C), mais aussi pour des langages plus récents tels que PHP ou Python. Ce terme, quelque peu abusif, pourrait laisser croire que les modules du C sont moins généraux que ceux des anciens langages. Or il n'en est rien, bien au contraire ! Certes, la fonction pourra y être utilisée comme dans ces langages, c'est-à-dire recevoir des arguments et fournir un résultat scalaire qu'on utilisera dans une expression, comme dans :

```
y = sqrt(x)+3 ;
```

Mais, en C, la fonction pourra prendre des aspects différents, pouvant complètement dénaturer l'idée que l'on se fait d'une fonction mathématique. Par exemple :

- La valeur d'une fonction pourra très bien ne pas être utilisée ; c'est ce qui se passe fréquemment lorsque vous employez `printf` ou `scanf`. Bien entendu, cela n'a d'intérêt que parce que de telles fonctions réalisent une **action**.
- Une fonction pourra ne fournir aucune valeur.
- Une fonction pourra ne posséder aucun argument.
- Une fonction pourra fournir un résultat non scalaire (nous n'en parlerons toutefois que dans le chapitre consacré aux structures).
- Une fonction pourra modifier les valeurs de certains de ses arguments (il vous faudra toutefois attendre d'avoir étudié les pointeurs pour voir par quel mécanisme elle y parviendra).

Par ailleurs, nous verrons qu'en C, plusieurs fonctions peuvent partager des informations, autrement que par passage d'arguments, à savoir par le biais de « variables globales ».

Enfin, l'un des atouts du langage C réside dans la possibilité de **compilation séparée**. Celle-ci permet de découper le programme source en plusieurs parties, chacune de ces parties pouvant comporter une ou plusieurs fonctions. Certains auteurs emploient parfois le mot « module » pour désigner chacune de ces parties (stockées dans un fichier) ; dans ce cas, ce terme de module devient synonyme de fichier source. Cela facilite considérablement le développement et la mise au point de grosses applications. Cette possibilité crée naturellement quelques contraintes supplémentaires, notamment au niveau des variables globales que l'on souhaite partager entre différentes parties du programme source (c'est d'ailleurs ce qui justifiera l'existence de la déclaration `extern`).

Pour garder une certaine progressivité dans notre exposé, nous supposerons tout d'abord que nous avons affaire à un programme source d'un seul tenant (ce qui ne nécessite donc pas de compilation séparée). Nous présenterons ainsi la structure générale d'une fonction, les notions d'arguments, de variables globales et locales. Ce n'est qu'alors que nous introduirons les possibilités de compilation séparée en montrant quelles sont ses incidences sur les points précédents ; cela nous amènera à parler des différentes « classes d'allocation » des variables.

2 Exemple de définition et d'utilisation d'une fonction en C

Nous vous proposons d'examiner tout d'abord un exemple simple de fonction correspondant à l'idée usuelle que l'on se fait d'une fonction, c'est-à-dire recevant des arguments et fournissant une valeur.

Exemple de définition et d'utilisation d'une fonction

```
#include <stdio.h>
          /***** le programme principal (fonction main) *****/
int main()
{
   float fexple (float, int, int) ; /* déclaration de fonction fexple */
   float x = 1.5 ;
   float y, z ;
   int n = 3, p = 5, q = 10 ;

         /* appel de fexple avec les arguments x, n et p */
   y = fexple (x, n, p) ;
   printf ("valeur de y : %e\n", y) ;
```

Exemple de définition et d'utilisation d'une fonction (suite)

```
        /* appel de fexple avec les arguments x+0.5, q et n-1 */
   z = fexple (x+0.5, q, n-1) ;
   printf ("valeur de z : %e\n", z) ;
}

        /*************** la fonction fexple ***************/
float fexple (float x, int b, int c)
{  float val ;          /* déclaration d'une variable "locale" à fexple
   val = x * x + b * x + c ;
   return val ;
}
```

Nous y trouvons tout d'abord, de façon désormais classique, un programme principal formé d'un bloc. Mais, cette fois, à sa suite, apparaît la **définition d'une fonction**. Celle-ci possède une structure voisine de la fonction `main`, à savoir un en-tête et un corps délimité par des accolades ({ et }). Mais l'en-tête est plus élaboré que celui de la fonction `main` puisque, outre le nom de la fonction (`fexple`), on y trouve une liste d'arguments (nom + type), ainsi que le type de la valeur qui sera fournie par la fonction (on la nomme indifféremment « résultat », « valeur de la fonction », « valeur de retour »...) :

```
float          fexple         (float x,        int b,        int c)
  |              |               |              |             |
type de la   nom de la        premier       deuxième      troisième
"valeur       fonction       argument       argument      argument
de retour"                (type float)   (type int)    (type int)
```

Les noms des arguments n'ont d'importance qu'au sein du corps de la fonction. Ils servent à décrire le travail que devra effectuer la fonction quand on l'appellera en lui fournissant trois valeurs.

Si on s'intéresse au corps de la fonction, on y rencontre tout d'abord une déclaration :

```
float val ;
```

Celle-ci précise que, pour effectuer son travail, notre fonction a besoin d'une variable de type `float` nommée `val`. On dit que `val` est une variable locale à la fonction `fexple`, de même que les variables telles que `n`, `p`, `y`... sont des variables locales à la fonction `main` (mais comme jusqu'ici nous avions affaire à un programme constitué d'une seule fonction, cette distinction n'était pas utile). Un peu plus loin, nous examinerons plus en détail cette notion de variable locale et celle de portée qui s'y attache.

L'instruction suivante de notre fonction `fexple` est une affectation classique (faisant toutefois intervenir les valeurs des arguments `x`, `n` et `p`).

Enfin, l'instruction `return val` précise la valeur que fournira la fonction à la fin de son travail.

En définitive, on peut dire que `fexple` est une fonction telle que `fexple (x, b, c)` fournisse la valeur de l'expression $x^2 + bx + c$. Notez bien l'aspect arbitraire du nom des arguments ; on obtiendrait la même définition de fonction avec, par exemple :

```
float fexple (float z, int coef, int n)
{
   float val ; /* déclaration d'une variable "locale" à fexple */
   val = z * z + coef * z + n ;
   return val ;
}
```

Examinons maintenant la fonction `main`. Vous constatez qu'on y trouve une déclaration :

```
   float fexple (float, int, int) ;
```

Elle sert à prévenir le compilateur que `fexple` est une fonction et elle lui précise le type de ses arguments ainsi que celui de sa valeur de retour. Nous reviendrons plus loin en détail sur le rôle d'une telle déclaration.

Quant à l'utilisation de notre fonction `fexple` au sein de la fonction `main`, elle est classique et comparable à celle d'une fonction prédéfinie telle que `scanf` ou `sqrt`. Ici, nous nous sommes contenté d'appeler notre fonction à deux reprises avec des arguments différents.

3 Quelques règles

3.1 Arguments muets et arguments effectifs

Les noms des arguments figurant dans l'en-tête de la fonction se nomment des « arguments muets », ou encore « arguments formels » ou « paramètres formels » (de l'anglais : *formal parameter)*. Leur rôle est de permettre, au sein du corps de la fonction, de décrire ce qu'elle doit faire.

Les arguments fournis lors de l'utilisation (l'appel) de la fonction se nomment des « arguments effectifs » (ou encore « paramètres effectifs »). Comme le laisse deviner l'exemple précédent, on peut utiliser n'importe quelle expression comme argument effectif ; au bout du compte, c'est la valeur de cette expression qui sera transmise à la fonction lors de son appel. Notez qu'une telle « liberté » n'aurait aucun sens dans le cas des paramètres formels : il serait impossible d'écrire un en-tête de `fexple` sous la forme `float fexple (float a+b, ...)` pas plus qu'en mathématiques vous ne définiriez une fonction f par $f(x+y)=5$!

3.2 L'instruction `return`

Voici quelques règles générales concernant cette instruction.

- L'instruction `return` peut mentionner n'importe quelle expression. Ainsi, nous aurions pu définir la fonction `fexple` précédente d'une manière plus simple :

```
float fexple (float x, int b, int c)
{
   return (x * x + b * x + c) ;
}
```

- L'instruction `return` peut apparaître à plusieurs reprises dans une fonction, comme dans cet autre exemple :

```
double absom (double u, double v)
{
    double s ;
    s = a + b ;
    if (s>0)    return (s) ;
         else   return (-s) ;
}
```

Notez bien que non seulement l'instruction `return` définit la valeur du résultat, mais, en même temps, elle interrompt l'exécution de la fonction en revenant dans la fonction qui l'a appelée (n'oubliez pas qu'en C tous les modules sont des fonctions, y compris le programme principal). Nous verrons qu'une fonction peut ne fournir aucune valeur : elle peut alors disposer de une ou plusieurs instructions `return` **sans expression**, interrompant simplement l'éxecution de la fonction ; mais elle peut aussi dans ce cas ne comporter aucune instruction `return`, le retour étant alors mis en place automatiquement par le compilateur à la fin de la fonction.

- Si le type de l'expression figurant dans `return` est différent du type du résultat tel qu'il a été déclaré dans l'en-tête, le compilateur mettra automatiquement en place des instructions de conversion.

  Il est toujours possible de ne pas utiliser le résultat d'une fonction, même si elle en produit un. C'est d'ailleurs ce que nous avons fait fréquemment avec `printf` ou `scanf`. Bien entendu, cela n'a d'intérêt que si la fonction fait autre chose que de calculer un résultat. En revanche, il est interdit d'utiliser la valeur d'une fonction ne fournissant pas de résultat (si certains compilateurs l'acceptent, vous obtiendrez, lors de l'exécution, une valeur aléatoire !).

3.3 Cas des fonctions sans valeur de retour ou sans arguments

Quand une fonction ne renvoie pas de résultat, on le précise, à la fois dans l'en-tête et dans sa déclaration, à l'aide du mot-clé `void`. Par exemple, voici l'en-tête d'une fonction recevant un argument de type `int` et ne fournissant aucune valeur :

```
void sansval (int n)
```

et voici quelle serait sa déclaration :

```
void sansval (int) ;
```

Naturellement, la définition d'une telle fonction ne doit, en principe, contenir aucune instruction `return`. Certains compilateurs ne détecteront toutefois pas l'erreur.

Quand une fonction ne reçoit aucun argument, on place le mot-clé `void` (le même que précédemment, mais avec une signification différente !) à la place de la liste d'arguments (attention, en C++, la règle sera différente : on se contentera de ne rien mentionner dans la liste d'arguments). Voici l'en-tête d'une fonction ne recevant aucun argument et renvoyant une valeur de type `float` (il pourrait s'agir, par exemple, d'une fonction fournissant un nombre aléatoire !) :

```
float tirage (void)
```

Sa déclaration serait très voisine (elle ne diffère que par la présence du point-virgule !) :

```
float tirage (void) ;
```

Enfin, rien n'empêche de réaliser une fonction ne possédant ni arguments ni valeur de retour. Dans ce cas, son en-tête sera de la forme :

```
void message (void)
```

et sa déclaration sera :

```
void message (void) ;
```

Voici un exemple illustrant deux des situations évoquées. Nous y définissons une fonction `affiche_carres` qui affiche les carrés des nombres entiers compris entre deux limites fournies en arguments et une fonction `erreur` qui se contente d'afficher un message d'erreur (il s'agit de notre premier exemple de programme source contenant plus de deux fonctions).

```
#include <stdio.h>
int main()
{  void affiche_carres (int, int) ; /* prototype de affiche_carres */
   void erreur (void) ;             /* prototype de erreur */
   int debut = 5, fin = 10 ;
      .....
   affiche_carres (debut, fin) ;
      .....
   if (...) erreur () ;
}
void affiche_carres (int d, int f)
{  int i ;
   for (i=d ; i<=f ; i++)
      printf ("%d a pour carré %d\n", i, i*i) ;
}
void erreur (void)
{  printf ("*** erreur ***\n") ; }
```

3.4 Les anciennes formes de l'en-tête des fonctions

Dans la première version du langage C, telle qu'elle a été définie par Kernighan et Ritchie, avant la normalisation par le comité ANSI, l'en-tête d'une fonction s'écrivait différemment de ce que nous avons vu ici. Par exemple, l'en-tête de notre fonction `fexple` aurait été :

```
float fexple (x, b, c)
float x ;
int b, c ;
```

La norme ANSI autorise les deux formes, lesquelles sont actuellement acceptées par la plupart des compilateurs. Toutefois, seule la forme moderne, c'est-à-dire celle que nous avons présentée précédemment, sera autorisée par C++.

Remarque

L'habitude veut que les en-têtes écrits sous l'ancienne forme le soient sur plusieurs lignes comme dans notre exemple. Mais rien ne nous empêcherait de l'écrire sous cette forme :

```
float fexple (x, b, c) float x ; int b, c ;
```

4 Les fonctions et leurs déclarations

4.1 Les différentes façons de déclarer (ou de ne pas déclarer) une fonction

Dans notre exemple du paragraphe 2, nous avions fourni la définition de la fonction `fexple` **après** celle de la fonction `main`. Mais nous aurions pu tout aussi bien faire l'inverse :

```
float fexple (float x, int b, int c)
{
    ....
}


int main()
{
   float fexple (float, int, int) ; /* déclaration de la fonc. fexple */
        .....
   y = fexple (x, n, p) ;
        .....
}
```

En toute rigueur, dans ce cas, la déclaration de la fonction `fexple` (ici, dans `main`) est facultative, car, lorsqu'il traduit la fonction `main`, le compilateur connaît déjà la fonction `fexple`. Néanmoins, nous vous déconseillons d'omettre la déclaration de `fexple` dans ce cas ; en effet, il est tout à fait possible qu'ultérieurement vous soyez amené à modifier votre programme source ou même à l'éclater en plusieurs fichiers source comme l'autorisent les possibilités de compilation séparée du langage C.

Par ailleurs, le langage C (mais pas le C++) vous permet d'effectuer des déclarations partielles en ne mentionnant pas le type des arguments ; ainsi, dans notre exemple du paragraphe 2, nous pourrions déclarer `fexple` de cette façon dans la fonction `main` :

```
float fexple () ;
```

Qui plus est, C vous autorise à ne pas déclarer du tout une fonction qui renvoie une valeur de type `int` (là encore, ce sera interdit en C++ ainsi qu'en C99).

Nous ne saurions trop vous conseiller d'éviter de telles possibilités. Toutefois, sachez que vous risquez d'employer la dernière sans y prendre garde. En effet, toute fonction que vous utiliserez sans l'avoir déclarée sera considérée par le compilateur comme ayant des arguments quelconques et fournissant un résultat de type `int`. Les conséquences en seront différentes suivant que ladite fonction est ou non fournie dans le même fichier source. Dans le premier cas, on obtiendra bien une erreur de compilation ; dans le second, en revanche, les conséquences n'apparaîtront (de manière plus ou moins voilée) que lors de l'exécution !

La déclaration complète d'une fonction porte le nom de **prototype**. Il est possible, dans un prototype, de faire figurer des noms d'arguments, lesquels sont alors totalement arbitraires ; cette possibilité a pour seul intérêt de pouvoir écrire des prototypes qui sont identiques à l'en-tête de la fonction (au point-virgule près), ce qui peut en faciliter la création automatique. Dans notre exemple du paragraphe 2, notre fonction `fexple` aurait pu être **déclarée** ainsi :

```
float fexple (float x, int b, int c) ;
```

4.2 Où placer la déclaration d'une fonction

La tendance la plus naturelle consiste à placer la déclaration d'une fonction à l'intérieur des déclarations de toute fonction l'utilisant ; c'est ce que nous avons fait jusqu'ici. Et, de surcroît, dans tous nos exemples précédents, la fonction utilisatrice était la fonction `main` elle-même ! Dans ces conditions, nous avions affaire à une déclaration locale dont la portée était limitée à la fonction où elle apparaissait.

Mais il est également possible d'utiliser des déclarations globales, en les faisant apparaître **avant la définition de la première fonction**. Par exemple, avec :

```
float fexple (float, int, int) ;
int main()
{  .....
}
void f1 (...)
{  .....
}
```

la déclaration de `fexple` est connue à la fois de `main` et de `f1`.

4.3 À quoi sert la déclaration d'une fonction

Nous avons vu que la déclaration d'une fonction est plus ou moins obligatoire et qu'elle peut être plus ou moins détaillée. Malgré tout, nous vous avons recommandé d'employer toujours la forme la plus complète possible qu'on nomme prototype. Dans ce cas, un tel prototype peut être utilisé par le compilateur, et cela de deux façons complètement différentes.

a) Si la définition de la fonction se trouve dans le même fichier source (que ce soit avant ou après la déclaration), il s'assure que les arguments muets ont bien le type défini dans le prototype. Dans le cas contraire, il signale une erreur.

b) Lorsqu'il rencontre un appel de la fonction, il met en place d'éventuelles conversions des valeurs des arguments effectifs dans le type indiqué dans le prototype. Par exemple, avec notre fonction `fexple` du paragraphe 2, un appel tel que :

```
fexple (n+1, 2*x, p)
```

sera traduit par :

- l'évaluation de la valeur de l'expression `n+1` (en `int`) et sa conversion en `float`,
- l'évaluation de la valeur de l'expression `2*x` (en `float`) et sa conversion en `int` ; il y a donc dans ce dernier cas une conversion dégradante.

Remarques

Rappelons que, lorsque le compilateur ne connaît pas le type des arguments d'une fonction, il utilise des règles de conversions systématiques : `char` *et* `short` -> `int` et `float` -> `double`. La fonction `printf` est précisément dans ce cas.

Compte tenu de la remarque précédente, seule une fonction déclarée avec un prototype pourra recevoir un argument de type `float`, `char` ou `short`.

5 Retour sur les fichiers en-tête

Nous avons déjà dit qu'il existe un certain nombre de fichiers d'extension `.h`, correspondant chacun à une classe de fonctions. On y trouve, entre autres choses, les prototypes de ces fonctions.

Ce point se révèle fort utile :

- d'une part pour effectuer des contrôles sur le nombre et le type des arguments mentionnés dans les appels de ces fonctions,
- d'autre part pour forcer d'éventuelles conversions auxquelles on risque de ne pas penser.

À titre d'illustration de ce dernier aspect, supposez que vous ayez écrit ces instructions :

```
float x, y ;
   .....
y = sqrt (x) ;
   .....
```

sans les faire précéder d'une quelconque directive `#include`.

Elles produiraient alors des **résultats faux**. En effet, il se trouve que la fonction `sqrt` s'attend à recevoir un argument de type `double` (ce qui sera le cas ici, compte tenu des conversions implicites), et elle fournit un résultat de type `double`. Or, lors de la traduction de votre programme, le compilateur ne le sait pas. Il attribue donc d'office à `sqrt` le type `int` et il met en place une conversion de la valeur de retour (laquelle sera en fait de type `double`) en `int`. On se trouve en présence des conséquences habituelles d'une mauvaise interprétation de type.

Un premier remède consiste à placer dans votre module la déclaration :

```
double sqrt(double) ;
```

mais encore faut-il que vous connaissiez de façon certaine le type de cette fonction.

Une meilleure solution consiste à placer, en début de votre programme, la directive :

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

laquelle incorporera automatiquement le prototype approprié (entre autres choses).

6 En C, les arguments sont transmis par valeur

Nous avons déjà eu l'occasion de dire qu'en C les arguments d'une fonction étaient transmis par valeur. Cependant, dans les exemples que nous avons rencontrés dans ce chapitre, les conséquences et les limitations de ce mode de transmission n'apparaissaient guère. Or voyez cet exemple :

Conséquences de la transmission par valeur des arguments

```
#include <stdio.h>
int main()
{  void echange (int a, int b) ;
   int n=10, p=20 ;
   printf ("avant appel   : %d %d\n", n, p) ;
   echange (n, p) ;
   printf ("après appel   : %d %d", n, p)
}
void echange (int a, int b)
{
   int c ;
   printf ("début echange : %d %d\n", a, b) ;
   c = a ;
   a = b ;
   b = c ;
   printf ("fin echange   : %d %d\n", a, b) ;
}
```

```
avant appel   : 10 20
début echange : 10 20
fin echange   : 20 10
après appel   : 10 20
```

La fonction `echange` reçoit deux valeurs correspondant à ses deux arguments muets `a` et `b`. Elle effectue un échange de ces deux valeurs. Mais, lorsque l'on est revenu dans le programme principal, aucune trace de cet échange ne subsiste sur les arguments effectifs `n` et `p`.

En effet, lors de l'appel de `echange`, il y a eu transmission de la valeur des expressions `n` et `p`. On peut dire que ces valeurs ont été recopiées localement dans la fonction `echange` dans

des emplacements nommés `a` et `b`. C'est effectivement sur ces copies qu'a travaillé la fonction `echange`, de sorte que les valeurs des variables `n` et `p` n'ont, quant à elles, pas été modifiées. C'est ce qui explique le résultat constaté.

Ce mode de transmission semble donc interdire a priori qu'une fonction produise une ou plusieurs valeurs en retour, autres que celle de la fonction elle-même.

Or, il ne faut pas oublier qu'en C tous les modules doivent être écrits sous forme de fonction. Autrement dit, ce simple problème d'échange des valeurs de deux variables doit pouvoir se résoudre à l'aide d'une fonction.

Nous verrons que ce problème possède plusieurs solutions, à savoir :

- Transmettre en argument la valeur de l'adresse d'une variable. La fonction pourra éventuellement agir sur le contenu de cette adresse. C'est précisément ce que nous faisons lorsque nous utilisons la fonction `scanf`. Nous examinerons cette technique en détail dans le chapitre consacré aux pointeurs.
- Utiliser des variables globales, comme nous le verrons dans le prochain paragraphe ; cette deuxième solution devra toutefois être réservée à des cas exceptionnels, compte tenu des risques qu'elle présente (effets de bords).

Remarques

C'est bien parce que la transmission des arguments se fait « par valeur » que les arguments effectifs peuvent prendre la forme d'une expression quelconque. Dans les langages où le seul mode de transmission est celui « par adresse », les arguments effectifs ne peuvent être que l'équivalent d'une *lvalue*.

La norme n'impose aucun ordre pour l'évaluation des différents arguments d'une fonction lors de son appel. En général, ceci est de peu d'importance, excepté dans une situation (fortement déconseillée !) telle que :

```
int i = 10 ;
   ...
f (i++, i) ;    /* on peut calculer i++ avant i --> f (10, 11) */
                /*                  ou après i --> f (10, 10) */
```

7 Les variables globales

Nous avons vu comment échanger des informations entre différentes fonctions grâce à la transmission d'arguments et à la récupération d'une valeur de retour.

En fait, en C, plusieurs fonctions (dont, bien entendu le programme principal `main`) peuvent partager des variables communes qu'on qualifie alors de **globales**.

Remarque

La norme ANSI ne parle pas de variables globales, mais de variables externes. Le terme « global » illustre plutôt le partage entre plusieurs fonctions tandis que le terme « externe » illustre plutôt le partage entre plusieurs fichiers source. En C, une variable globale est partagée par plusieurs fonctions ; elle peut être (mais elle n'est pas obligatoirement) partagée entre plusieurs fichiers source.

7.1 Exemple d'utilisation de variables globales

Voyez cet exemple de programme.

Exemple d'utilisation d'une variable globale

```
#include <stdio.h>
int i ;
int main()
{  void optimist (void) ;
   for (i=1 ; i<=5 ; i++)
       optimist() ;
}
void optimist(void)
{   printf ("il fait beau %d fois\n", i) ;
}
```

```
il fait beau 1 fois
il fait beau 2 fois
il fait beau 3 fois
il fait beau 4 fois
il fait beau 5 fois
```

La variable `i` a été déclarée en dehors de la fonction `main`. Elle est alors connue de toutes les fonctions qui seront compilées par la suite au sein du même programme source. Ainsi, ici, le programme principal affecte à `i` des valeurs qui se trouvent utilisées par la fonction `optimist`.

Notez qu'ici la fonction `optmist` se contente d'utiliser la valeur de `i` mais rien ne l'empêche de la modifier. C'est précisément ce genre de remarque qui doit vous inciter à n'utiliser les variables globales que dans des cas limités. En effet, toute variable globale peut être modifiée insidieusement par n'importe quelle fonction. Lorsque vous aurez à écrire des fonctions susceptibles de modifier la valeur de certaines variables, il sera beaucoup plus judicieux de prévoir d'en transmettre l'adresse en argument (comme vous apprendrez à le faire dans le prochain chapitre). En effet, dans ce cas, l'appel de la fonction montrera explicitement quelle est la variable qui risque d'être modifiée et, de plus, ce sera la seule qui pourra l'être.

7.2 La portée des variables globales

Les variables globales ne sont connues du compilateur que dans la partie du programme source suivant leur déclaration. On dit que leur **portée** (ou encore leur **espace de validité**) est limitée à la partie du programme source qui suit leur déclaration (n'oubliez pas que, pour l'instant, nous nous limitons au cas où l'ensemble du programme est compilé en une seule fois).

Ainsi, voyez, par exemple, ces instructions :

```
int main()
{
   ....
}
int n ;
float x ;
fct1 (...)
{
   ....
}
fct2 (...)
{
   ....
}
```

Les variables `n` et `x` sont accessibles aux fonctions `fct1` et `fct2`, mais pas au programme principal. En pratique, bien qu'il soit possible effectivement de déclarer des variables globales à n'importe quel endroit du programme source qui soit extérieur aux fonctions, on procédera rarement ainsi. En effet, pour d'évidentes raisons de lisibilité, on préférera regrouper les déclarations de toutes les variables globales au début du programme source.

7.3 La classe d'allocation des variables globales

D'une manière générale, les variables globales existent pendant toute l'exécution du programme dans lequel elles apparaissent. Leurs emplacements en mémoire sont parfaitement définis lors de l'édition de liens. On traduit cela en disant qu'elles font partie de la **classe d'allocation statique**.

De plus, ces variables se voient **initialisées à zéro**, avant le début de l'exécution du programme, sauf, bien sûr, si vous leur attribuez explicitement une valeur initiale au moment de leur déclaration.

8 Les variables locales

À l'exception de l'exemple du paragraphe précédent, les variables que nous avions rencontrées jusqu'ici n'étaient pas des variables globales. Plus précisément, elles étaient définies au sein d'une fonction (qui pouvait être `main`). De telles variables sont dites **locales** à la fonction dans laquelle elles sont déclarées.

8.1 La portée des variables locales

Les variables locales ne sont connues qu'à l'intérieur de la fonction où elles sont déclarées. **Leur portée est donc limitée à cette fonction.**

Les variables locales n'ont aucun lien avec des variables globales de même nom ou avec d'autres variables locales à d'autres fonctions.

Voyez cet exemple :

```
int n ;
int main()
{
   int p ;
   ....
}
fct1 ()
{
   int p ;
   int n ;
}
```

La variable `p` de `main` n'a aucun rapport avec la variable `p` de `fct1`. De même, la variable `n` de `fct1` n'a aucun rapport avec la variable globale `n`. Notez qu'il est alors impossible, dans la fonction `fct1`, d'utiliser cette variable globale `n`.

8.2 Les variables locales automatiques

Par défaut, les variables locales ont une durée de vie limitée à celle d'**une exécution** de la fonction dans laquelle elles figurent.

Plus précisément, leurs emplacements ne sont pas définis de manière permanente comme ceux des variables globales. Un nouvel espace mémoire leur est alloué à chaque entrée dans la fonction et libéré à chaque sortie. Il sera donc généralement différent d'un appel au suivant.

On traduit cela en disant que la **classe d'allocation** de ces variables est **automatique**. Nous aurons l'occasion de revenir plus en détail sur cette gestion dynamique de la mémoire. Pour l'instant, il est important de noter que la conséquence immédiate de ce mode d'allocation est que les valeurs des variables locales ne sont pas conservées d'un appel au suivant (on dit aussi qu'elles ne sont pas « rémanentes »). Nous reviendrons un peu plus loin (paragraphe 11.2) sur les éventuelles initialisations de telles variables.

D'autre part, les valeurs transmises en arguments à une fonction sont traitées de la même manière que les variables locales. Leur durée de vie correspond également à celle de la fonction.

8.3 Les variables locales statiques

Il est toutefois possible de demander d'attribuer un emplacement permanent à une variable locale et qu'ainsi sa valeur se conserve d'un appel au suivant. Il suffit pour cela de la déclarer à l'aide du mot-clé **`static`** (le mot `static` employé sans indication de type est équivalent à `static int`).

En voici un exemple :

Exemple d'utilisation de variable locale statique

```
#include <stdio.h>
int main()
{  void fct(void) ;
   int n ;
   for ( n=1 ; n<=5 ; n++)
      fct() ;
}
void fct(void)
{  static int i ;
   i++ ;
   printf ("appel numéro : %d\n", i) ;
}
```

```
appel numéro : 1
appel numéro : 2
appel numéro : 3
appel numéro : 4
appel numéro : 5
```

La variable locale `i` a été déclarée de classe « statique ». On constate bien que sa valeur progresse de un à chaque appel. De plus, on note qu'au premier appel sa valeur est nulle. En effet, comme pour les variables globales (lesquelles sont aussi de classe statique) : **les variables locales de classe statique sont, par défaut, initialisées à zéro**.

Prenez garde à ne pas confondre une variable locale de classe statique avec une variable globale. En effet, la portée d'une telle variable reste toujours limitée à la fonction dans laquelle elle est définie. Ainsi, dans notre exemple, nous pourrions définir une variable globale nommée `i` qui n'aurait alors aucun rapport avec la variable `i` de `fct`.

8.4 Le cas des fonctions récursives

Le langage C autorise la récursivité des appels de fonctions. Celle-ci peut prendre deux aspects :

- récursivité directe : une fonction comporte, dans sa définition, au moins un appel à elle-même,
- récursivité croisée : l'appel d'une fonction entraîne celui d'une autre fonction qui, à son tour, appelle la fonction initiale (le cycle pouvant d'ailleurs faire intervenir plus de deux fonctions).

Voici un exemple fort classique (d'ailleurs inefficace sur le plan du temps d'exécution) d'une fonction calculant une factorielle de manière récursive :

Fonction récursive de calcul de factorielle

```
long fac (int n)
{
   if (n>1) return (fac(n-1)*n) ;
       else return(1) ;
}
```

Il faut bien voir qu'alors chaque appel de `fac` entraîne une allocation d'espace pour les variables locales et pour son argument `n` (apparemment, `fct` ne comporte aucune variable locale ; en réalité, il lui faut prévoir un emplacement destiné à recevoir sa valeur de retour). Or chaque nouvel appel de `fac`,à l'intérieur de `fac`, provoque une telle allocation, sans que les emplacements précédents soient libérés.

Il y a donc un empilement des espaces alloués aux variables locales, parallèlement à un empilement des appels de la fonction. Ce n'est que lors de l'exécution de la première instruction `return` que l'on commencera à « dépiler » les appels et les emplacements et donc à libérer de l'espace mémoire.

9 La compilation séparée et ses conséquences

Si le langage C est effectivement un langage que l'on peut qualifier d'opérationnel, c'est en partie grâce à ses possibilités dites de **compilation séparée**. En C, en effet, il est possible de compiler séparément plusieurs programmes (fichiers) source et de rassembler les modules objet correspondants au moment de l'édition de liens. D'ailleurs, dans certains environnements de programmation, la notion de *projet* permet de gérer la multiplicité des fichiers (source et modules objet) pouvant intervenir dans la création d'un programme exécutable. Cette notion de projet fait intervenir précisément les fichiers à considérer ; généralement, il est possible de demander de créer le programme exécutable, en ne recompilant que les sources ayant subi une modification depuis leur dernière compilation.

Indépendamment de ces aspects techniques liés à l'environnement de programmation considéré, les possibilités de compilation séparée ont une incidence importante au niveau de la portée des variables globales. C'est cet aspect que nous nous proposons d'étudier maintenant. Dans le

paragraphe suivant, nous serons alors en mesure de faire le point sur les différentes classes d'allocation des variables.

Notez que, à partir du moment où l'on parle de compilation séparée, il existe au moins (ou il a existé) deux programmes source ; dans la suite, nous supposerons qu'ils figurent dans des fichiers, de sorte que nous parlerons toujours de fichier source.

Pour l'instant, voyons l'incidence de cette compilation séparée sur la portée des variables globales.

9.1 La portée d'une variable globale - la déclaration extern

A priori, la portée d'une variable globale semble limitée au fichier source dans lequel elle a été définie. Ainsi, supposez que l'on compile séparément ces deux fichiers source :

```
source 1                    source 2
 int x ;                      fct2()
 int main()                   {
 {                                .....
    .....                     }
 }                            fct3()
 fct1()                       {
 {                                .....
    .....                     }
 }
```

Il ne semble pas possible, dans les fonctions `fct2` et `fct3` de faire référence à la variable globale `x` déclarée dans le premier fichier source (alors qu'aucun problème ne se poserait si l'on réunissait ces deux fichiers source en un seul, du moins si l'on prenait soin de placer les instructions du second fichier à la suite de celles du premier).

En fait, le langage C prévoit une déclaration permettant de spécifier qu'une variable globale a déjà été définie dans un autre fichier source. Celle-ci se fait à l'aide du mot-clé `extern`. Ainsi, en faisant précéder notre second fichier source de la déclaration :

```
extern int x ;
```

il devient possible de mentionner la variable globale `x` (déclarée dans le premier fichier source) dans les fonctions `fct2` et `fct3`.

Remarques

Cette déclaration `extern` n'effectue **pas de réservation d'emplacement de variable**. Elle ne fait que préciser que la variable globale `x` est définie par ailleurs et elle en précise le type.

Nous n'avons considéré ici que la façon la plus usuelle de gérer des variables globales (celle-ci est utilisable avec tous les compilateurs, qu'ils soient d'avant ou d'après la norme). La norme prévoit d'autres possibilités, au demeurant fort peu répandues et, de surcroît, peu logiques (doubles déclarations, mot `extern` utilisé même pour la réservation d'un emplacement, définitions potentielles).

9.2 Les variables globales et l'édition de liens

Supposons que nous ayons compilé les deux fichiers source précédents et voyons d'un peu plus près comment l'éditeur de liens est en mesure de rassembler correctement les deux modules objet ainsi obtenus. En particulier, examinons comment il peut faire correspondre au symbole `x` du second fichier source l'adresse effective de la variable `x` définie dans le premier.

D'une part, après compilation du premier fichier source, on trouve, dans le module objet correspondant, une indication associant le symbole `x` et son adresse dans le module objet. Autrement dit, contrairement à ce qui se produit pour les variables locales, pour lesquelles ne subsiste aucune trace du nom après compilation, le nom des variables globales continue à exister au niveau des modules objet. On retrouve là un mécanisme analogue à ce qui se passe pour les noms de fonctions, lesquels doivent bien subsister pour que l'éditeur de liens soit en mesure de retrouver les modules objet correspondants.

D'autre part, après compilation du second fichier source, on trouve dans le module objet correspondant, une indication mentionnant qu'une certaine variable de nom `x` provient de l'extérieur et qu'il faudra en fournir l'adresse effective.

Ce sera effectivement le rôle de l'éditeur de liens que de retrouver dans le premier module objet l'adresse effective de la variable `x` et de la reporter dans le second module objet.

Ce mécanisme montre que s'il est possible, par mégarde, de réserver des variables globales de même nom dans deux fichiers source différents, il sera, par contre, en général, impossible d'effectuer correctement l'édition de liens des modules objet correspondants (certains éditeurs de liens peuvent ne pas détecter cette anomalie). En effet, dans un tel cas, l'éditeur de liens se trouvera en présence de deux adresses différentes pour un même identificateur, ce qui est illogique.

9.3 Les variables globales cachées - la déclaration static

Il est possible de « cacher » une variable globale, c'est-à-dire de la rendre inaccessible à l'extérieur du fichier source où elle a été définie (on dit aussi « rendre confidentielle » au lieu de « cacher » ; on parle alors de « variables globales confidentielles »). Il suffit pour cela d'utiliser la déclaration **`static`** comme dans cet exemple :

```
static int a ;
int main()
{
   .....
}
fct()
{
   .....
}
```

Sans la déclaration `static`, `a` serait une variable globale ordinaire. Par contre, cette déclaration demande qu'aucune trace de `a` ne subsiste dans le module objet résultant de ce fichier source. Il sera donc impossible d'y faire référence depuis une autre source par `extern`. Mieux, si une autre variable globale apparaît dans un autre fichier source, elle sera acceptée à l'édition de liens puisqu'elle ne pourra pas interférer avec celle du premier source.

Cette possibilité de cacher des variables globales peut s'avérer précieuse lorsque vous êtes amené à développer un ensemble de fonctions d'intérêt général qui doivent partager des variables globales. En effet, elle permet à l'utilisateur éventuel de ces fonctions de ne pas avoir à se soucier des noms de ces variables globales puisqu'il ne risque pas alors d'y accéder par mégarde. Cela généralise en quelque sorte à tout un fichier source la notion de variable locale telle qu'elle était définie pour les fonctions. Ce sont d'ailleurs de telles possibilités qui permettent de développer des logiciels en C, en utilisant une philosophie orientée objet.

10 Les différents types de variables, leur portée et leur classe d'allocation

Nous avons déjà vu différentes choses concernant les classes d'allocation des variables et leur portée. Ici, nous nous proposons de faire le point après avoir introduit quelques informations supplémentaires.

10.1 La portée des variables

On peut classer les variables en quatre catégories en fonction de leur portée (ou espace de validité). Nous avons déjà rencontré les trois premières que sont : les variables globales, les variables globales cachées et les variables locales à une fonction. En outre, il est possible de définir des **variables locales à un bloc**. Elles se déclarent en début d'un bloc de la même façon qu'en début d'une fonction. Dans ce cas, la portée de telles variables est limitée au bloc en question. Leur usage est, en pratique, peu répandu.

10.2 Les classes d'allocation des variables

Il est également possible de classer les variables en trois catégories en fonction de leur classe d'allocation. Là encore, nous avons déjà rencontré les deux premières, à savoir :

- **la classe statique** : elle renferme non seulement les variables globales (quelles qu'elles soient), mais aussi les variables locales faisant l'objet d'une déclaration `static`. Les emplacements mémoire correspondants sont alloués une fois pour toutes au moment de l'édition de liens,

- **la classe automatique** : par défaut, les variables locales entrent dans cette catégorie. Les emplacements mémoire correspondants sont alloués à chaque entrée dans la fonction où sont définies ces variables et ils sont libérés à chaque sortie.

En toute rigueur, il existe une classe un peu particulière, à savoir la **classe registre :** toute variable entrant a priori dans la classe automatique peut être déclarée explicitement avec le qualificatif **`register`**. Celui-ci demande au compilateur d'utiliser, dans la mesure du possible, un registre pour y ranger la variable ; cela peut amener quelques gains de temps d'exécution. Bien entendu, cette possibilité ne peut s'appliquer qu'aux variables scalaires.

Remarque

Le cas des fonctions. La fonction est considérée par le langage C comme un objet global. C'est ce qui permet d'ailleurs à l'éditeur de liens d'effectuer correctement son travail. Il faut noter toutefois qu'il n'est pas nécessaire d'utiliser une déclaration `extern` pour les fonctions définies dans un fichier source différent de celui où elles sont appelées (mais le faire ne constitue pas une erreur).

En tant qu'objet global, la fonction peut voir sa portée limitée au fichier source dans lequel elle est définie à l'aide d'une déclaration `static` comme dans :

```
static int fct (...)
```

10.3 Tableau récapitulatif

Voici un tableau récapitulant la portée et la classe d'allocation des différentes variables suivant la nature de leur déclaration (la colonne « Type » donne le nom qu'on attribue usuellement à de telles variables).

Type, portée et classe d'allocation des variables

Type de variable	Déclaration	Portée	Classe d'allocation
Globale	en dehors de toute fonction	• la partie du fichier source suivant sa déclaration, • n'importe quel fichier source, avec `extern`.	**Statique**
Globale cachée	en dehors de toute fonction, avec l'attribut `static`	uniquement la partie du fichier source suivant sa déclaration	
Locale « rémanente »	au début d'une fonction, avec l'attribut `static`	la fonction	
Locale à une fonction	au début d'une fonction	la fonction	**Automatique**
Locale à un bloc	au début d'un bloc	le bloc	

11 Initialisation des variables

Nous avons vu qu'il était possible d'initialiser explicitement une variable lors de sa déclaration. Ici, nous allons faire le point sur ces possibilités, lesquelles dépendent en fait de la classe d'allocation de la variable concernée.

11.1 Les variables de classe statique

Ces variables sont permanentes. Elles sont initialisées une seule fois avant le début de l'exécution du programme.

Elles peuvent être initialisées explicitement lors de leur déclaration. Nous verrons que cela s'applique également aux tableaux ou aux structures. Bien entendu, les valeurs servant à ces initialisations ne pourront être que des constantes ou des expressions constantes (c'est-à-dire calculables par le compilateur). N'oubliez pas que les constantes symboliques définies par `const` ne sont pas considérées comme des expressions constantes.

En l'absence d'initialisation explicite, ces variables seront initialisées à zéro.

11.2 Les variables de classe automatique

Ces variables ne sont pas initialisées par défaut. En revanche, comme les variables de classe statique, elles peuvent être initialisées explicitement lors de leur déclaration.

Dans ce cas, lorsqu'il s'agit de variables scalaires (ce qui est le cas de toutes celles rencontrées jusqu'ici, mais ne sera pas le cas des tableaux ou des structures), la norme vous autorise à utiliser comme valeur initiale non seulement une valeur constante, mais également n'importe quelle expression (dans la mesure où sa valeur est définie au moment de l'entrée dans la fonction correspondante, laquelle, ne l'oubliez pas, peut être la fonction `main`).

En voici un cas d'école :

Initialisation de variables de classe automatique

```
#include <stdio.h>
int n ;
int main()
{ void fct (int r) ;
  int p ;
  for (p=1 ; p<=5 ; p++)
    { n = 2*p ;
      fct(p) ;
    }
}
void fct(int r)
{
  int q=n, s=r*n ;
  printf ("%d %d %d\n", r,q,s) ;
}
```

N'oubliez pas que ces variables automatiques se trouvent alors initialisées à chaque appel de la fonction dans laquelle elles sont définies.

12 Les arguments variables en nombre

Dans tous nos précédents exemples, le nombre d'arguments fournis au cours de l'appel d'une fonction était prévu lors de l'écriture de cette fonction.

Or, dans certaines circonstances, on peut souhaiter réaliser une fonction capable de recevoir un nombre d'arguments susceptible de varier d'un appel à un autre. C'est d'ailleurs ce que nous faisons (sans même y penser) lorsque nous utilisons des fonctions comme `printf` ou `scanf` (en dehors du premier argument, qui représente le format, les autres sont en nombre quelconque)

Le langage C permet de résoudre ce problème à l'aide des fonctions particulières `va_start` et `va_arg`. La seule contrainte à respecter est que la fonction doit posséder certains arguments fixes (c'est-à-dire toujours présents), leur nombre ne pouvant être inférieur à un. En effet, comme nous allons le voir, c'est le dernier argument fixe qui permet, en quelque sorte, d'initialiser le parcours de la liste d'arguments.

12.1 Premier exemple

Voici un premier exemple de fonction à arguments variables : les deux premiers arguments sont fixes, l'un étant de type `int`, l'autre de type `char`. Les arguments suivants, de type `int`, sont en nombre quelconque et l'on a supposé que le dernier d'entre eux était `-1`. Cette dernière valeur sert donc, en quelque sorte, de sentinelle. Par souci de simplification, nous nous sommes contentés, dans cette fonction, de lister les valeurs de ces différents arguments (fixes ou variables), à l'exception du dernier.

Arguments en nombre variable délimités par une sentinelle

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

void essai (int par1, char  par2, ...)
{
   va_list adpar ;
   int parv ;
   printf ("premier paramètre : %d\n", par1) ;
   printf ("second paramètre  : %c\n", par2) ;
   va_start (adpar, par2) ;
   while ( (parv = va_arg (adpar, int) ) != -1)
       printf ("argument variable : %d\n", parv) ;
}
```

Arguments en nombre variable délimités par une sentinelle (suite)

```
int main()
{
   printf ("premier essai\n") ;
   essai (125, 'a', 15, 30, 40, -1) ;
   printf ("\ndeuxième essai\n") ;
   essai (6264, 'S', -1) ;
}
```

```
premier essai
premier paramètre : 125
second paramètre  : a
argument variable : 15
argument variable : 30
argument variable : 40

deuxième essai
premier paramètre : 6264
second paramètre  : S
```

Vous constatez la présence, dans l'en-tête, des deux noms des paramètres fixes `par1` et `par2`, déclarés de manière classique ; les trois points servent à spécifier au compilateur l'existence de paramètres en nombre variable.

La déclaration :

```
va_list adpar
```

précise que `adpar` est un identificateur de liste variable. C'est lui qui nous servira à récupérer, les uns après les autres, les différents arguments variables.

Comme à l'accoutumée, une telle déclaration n'attribue aucune valeur à `adpar`. C'est effectivement la fonction `va_start` qui va permettre de l'initialiser à l'adresse du paramètre variable. Notez bien que cette dernière est déterminée par `va_start` à partir de la connaissance du nom du dernier paramètre fixe.

Le rôle de la fonction `va_arg` est double :

- d'une part, elle fournit comme résultat la valeur trouvée à l'adresse courante fournie par `adpar` (son premier argument), suivant le type indiqué par son second argument (ici `int`).
- d'autre part, elle incrémente l'adresse contenue dans `adpar`, de manière que celle-ci pointe alors sur l'argument variable suivant.

Ici, une instruction `while` nous permet de récupérer les différents arguments variables, sachant que le dernier a pour valeur `-1`.

Enfin, la norme ANSI prévoit que la macro `va_end` doit être appelée avant de sortir de la fonction concernée. Si vous manquez à cette règle, vous courrez le risque de voir un prochain appel à la fonction conduire à un mauvais fonctionnement du programme.

Remarques

Les arguments variables peuvent être de types différents, à condition toutefois que la fonction soit en mesure de les connaître, d'une façon ou d'une autre.

Les macros `va_start` et `va_end`, ainsi que la description du type `va_list`, figurent dans le fichier en-tête `stdarg.h` (d'où la directive `#include` correspondante). Cette description utilise une méthode de définition de type (instruction `typedef`) qui ne sera exposée que dans le chapitre relatif aux structures. Notez bien qu'ici vous n'avez pas besoin de savoir ce qui se cache derrière `va_list` pour l'utiliser correctement.

12.2 Second exemple

La gestion de la fin de la liste des arguments variables est laissée au bon soin de l'utilisateur ; en effet, il n'existe aucune fonction permettant de connaître le nombre effectif de ces arguments.

Cette gestion peut se faire :

- par sentinelle, comme dans notre précédent exemple,
- par transmission, en argument fixe, du nombre d'arguments variables.

Voici un exemple de fonction utilisant cette seconde technique. Nous n'y avons pas prévu d'autres arguments fixes que celui spécifiant le nombre d'arguments variables.

Arguments variables dont le nombre est fourni en argument fixe

```
#include <stdio.h>
#include <stdarg.h>
void essai (int nbpar, ...)
{  va_list adpar ;
   int parv, i ;
   printf ("nombre de valeurs : %d\n", nbpar) ;
   va_start (adpar, nbpar) ;
   for (i=1 ; i<=nbpar ; i++)
     {  parv = va_arg (adpar, int) ;
        printf ("argument variable : %d\n", parv) ;
     }
}
```

Arguments variables dont le nombre est fourni en argument fixe (suite)

```
int main()
{  printf ("premier essai\n") ;
   essai (3, 15, 30, 40) ;
   printf ("\ndeuxième essai\n") ;
   essai (0) ;
}
```

```
premier essai
nombre de valeurs : 3
argument variable : 15
argument variable : 30
argument variable : 40

deuxième essai
nombre de valeurs : 0
```

13 Cas particulier de la fonction main

Jusqu'ici, nous avons écrit de cette manière l'en-tête de la fonction *main* :

```
int main()
```

Effectivement, cette fonction dispose d'une valeur de retour de type `int`, dont la valeur peut être fixée à l'aide d'une classique instruction `return`. Lorsqu'elle existe, cette valeur peut être utilisée par l'environnement dans lequel s'est exécuté votre programme (mais cela n'est pas certain). Il est d'usage de renvoyer la valeur 0 pour indiquer qu'un programme s'est bien déroulé et une valeur autre dans le cas contraire.

Notez que, en toute rigueur, cet en-tête devrait s'écrire (comme pour toute fonction ne recevant aucun argument) :

```
int main(void)
```

Toutefois, la forme que nous utilisons reste la plus répandue (et ce sera celle de C++).

Par ailleurs, comme nous le verrons au paragraphe 11.2 du chapitre 8, il existe une deuxième forme d'en-tête, permettant à la fonction *main* de récupérer les valeurs d'arguments qu'on lui fournit avant son exécution.

Exercices

Tous ces exercices sont corrigés en fin de volume.

1) Écrire :

- une fonction, nommée `f1`, se contentant d'afficher `"bonjour"` (elle ne possèdera aucun argument ni valeur de retour),
- une fonction, nommée `f2`, qui affiche `"bonjour"` un nombre de fois égal à la valeur reçue en argument (`int`) et qui ne renvoie aucune valeur,
- une fonction, nommée `f3`, qui fait la même chose que `f2`, mais qui, de plus, renvoie la valeur (`int`) `0`.

Écrire un petit programme appelant successivement chacune de ces trois fonctions, après les avoir convenablement déclarées sous forme d'un prototype.

2) Qu'affiche le programme suivant ?

```
int n=5 ;
int main()
{
   void fct (int p) ;
   int n=3 ;
   fct(n) ;
}
void fct(int p)
{
   printf("%d %d", n, p) ;
}
```

3) Écrire une fonction qui se contente de comptabiliser le nombre de fois où elle a été appelée en affichant seulement un message de temps en temps, à savoir :

- au premier appel : *** appel 1 fois ***
- au dixième appel : *** appel 10 fois ***
- au centième appel : *** appel 100 fois ***
- et ainsi de suite pour le millième, le dix millième appel...
- On supposera que le nombre maximal d'appels ne peut dépasser la capacité d'un `long`.

4) Écrire une fonction récursive calculant la valeur de la « fonction d'Ackermann » `A` définie pour `m>0` et `n>0` par :

```
A(m,n) = A(m-1,A(m,n-1))     pour m>0 et n>0
A(0,n) = n+1     pour n>0
A(m,0) = A(m-1,1)     pour m>0
```