MAP C

Introduction à la programmation en C

Antoine Cornuéjols

AgroParisTech (2006-2007)

Version (très provisoire) du 17 mai 2007

Avertissement

Remerciements

Ce polycopié doit beaucoup aux nombreux ouvrages excellents qui décrivent le langage C. Il doit particulièrement aux documents mis en libre accès sur Internet. Je citerais en particulier :

- le cours d'Anne Canteaut (`http ://www-rocq.inria.fr/codes/Anne.Canteaut/COURS_C/`)
- le cours de Bernard Cassagne (`http ://clips.imag.fr/commun/bernard.cassagne/Introduction_ANSI_C`)

que j'ai pillés allègrement.

Je tiens à remercier tout spécialement mes étudiants du cours de MAP C à AgroParisTech qui ont subi ce cours donné pour la première fois en 2007. Leurs très nombreuses questions et suggestions m'ont considérablement aidé à organiser le matériel du cours et du polycopié. Naturellement, les erreurs et approximations, qui sans doute l'émaillent encore, me sont entièrement imputables.

Merci donc à : *Camille Amosse, Erwann Arc, Bénédicte Bachelot, Perrine Barillet, Vincent Briot, Marion Charme, Marie Cognet, Etienne Damerose, Camille Delebecque, Aude Deloumeau, Sylvain Du Peloux De Saint Romain, Florianne François, Axel Gamburzew, Aude Garda, Jean-Philippe Krieger, Denis Maret, Elen Rondel, Clémentine Roux, Pierre Scemama, et Hélène Schernberg.*

L'adresse Web : `http ://www.lri.fr/ antoine/Courses/AGRO/MAP-C/page-agro-map-C.html` contient la page d'accueil du cours MAP C..

Table des matières

Chapitre 1

Concepts fondamentaux

Sommaire

1 Introduction

1.1 Concepts fondamentaux en informatique

1.1.1 Système d'exploitation

L'ensemble des programmes informatiques qui contrôlent les interactions entre l'utilisateur et le système est appelé **système d'exploitation** (*ES*) (*Operating System*, en aglais, ou *OS*). Le système d'exploitation d'un ordinateur est souvent comparé à un chef d'orchestre car c'est le logiciel responsable de la direction de toutes les opérations et de la gestion des ressources.

Généralement, une partie au moins du système d'exploitation est stocké en permanence dans une mémoire en lecture seule (ROM : *Read-Only Memory*) afin d'être immédiatement disponible dès que l'ordinateur est allumé. Cette partie de l'*ES* contient les instructions nécessaires pour aller chercher le reste du système d'exploitation en mémoire, qui le plus souvent réside sur un disque. Le chargement du système d'exploitation s'appelle le *booting*.

Une liste des responsabilités d'un système d'exploitation inclut :
- Communication avec l'utilisateur : réception des commandes et exécution, avec éventuellement émission de messages d'erreur.
- Gestion des allocations mémoire, du temps de processeur, et des autres ressources pour les différentes tâches.
- Collecte des entrées à partir du clavier, de la souris et d'autres capteurs, et transmission aux programmes en cours d'exécution.
- Transmission des sorties vers écran(s), impirmante(s) et autres dispositifs de sorties.
- Lecture de données à partir de dispositifs de stockage secondaires (e.g. disques durs).
- Écriture de données sur dispositifs de stockage secondaires (e.g. disques durs).

Dans le cas de systèmes multi-utilisateurs, le système d'exploitation doit en plus vérifier les autorisations des utilisateurs.

La table suivante donne une liste de systèmes d'exploitation répandus.

Interface à ligne de commande	**Interface graphique**
`UNIX`	`Macintosh OS`
`MS-DOS`	`Windows`
`VMS`	`OS/2 Warp`
	`UNIX + X Windos system`

1.1.2 Logiciels applicatifs

Les logiciels applicatifs sont développés pour offrir une aide aux utilisateurs pour des tâches déterminées, comme par exemple le traitement de texte, le calcul dans des tableurs, la gestion de bases de données, ou des tâches de calcul formel ou d'analyse de molécules chimiques.

1.1.3 Niveaux de langages

1. `Langage machine`. Code à base de nombres binaires compréhensibles par des CPU spécifiques.
2. `Langage assembleur`. Codes mnémoniques correspondant à des instructions machine élémentaires.
3. `Langage de "haut-niveau"`. Langage indépendant d'une machine et qui permet de faire des opérations complexes à partir d'instructions simples.

4. `Compilateur`. Programme qui traduit un programme exprimé en langage de haut-niveau (*fichier source*, e.g. `myprog.c`) sous forme de langage machine (*fichier objet*, e.g. `myprog.obj`), c'est-à-dire sous la forme d'un fichier binaire.

Rôle du linker.

Bien qu'un fichier objet contienne des instructions machine, il peut ne pas être complet et suffisant pour l'exécution d'un programme. En effet, le programme écrit en langage de haut-niveau peut faire appel à des fonctions prédéfinies dans des librairies de programme disponibles par ailleurs. Le *linker* est un programme qui combine les informations de ces librairies et du programme à compiler pour produire un code machine exécutable. Cela peut résulter par exemple dans un programme `myprog.exe`.

1.1.4 Environnement de développement

Il existe des environnements de programmation qui sont des packages offrant les services d'un éditeur de texte, d'un compilateur, d'un linker et d'un exécuteur sous une interface intégrée. (Exemple : *Eclipse, Dev-C++*).

1.1.5 Exécution d'un programme

[HK], ch.1, p.20

Pas nécessaire.

1.2 Historique des langages de programmation

1. Machine de Pascal
2. The differencial engine
3. Machine de Turing
4. Architecture de von Neuman
5. Premiers langages : très proches des langages machines
6. Premiers langages de haut-niveau : FORTRAN et LISP
7. PASCAL, BASIC, C, ADA, SMALLTALK, FORTH, JAVA

1.3 Historique du langage C

Dans les dernières années, aucun langage de programmation n'a pu se vanter d'une croissance en popularité comparable à celle de C et de son jeune frère C++. L'étonnant dans ce fait est que le langage C n'est pas un nouveau-né dans le monde informatique, mais qu'il trouve ses sources en 1972 dans les 'Bell Laboratories' : Pour développer une version portable du système d'exploitation UNIX, Dennis M. Ritchie a conçu ce langage de programmation structuré, mais très 'près' de la machine.

Le C a été conçu en 1972 par Dennis Richie et Ken Thompson, chercheurs aux Bell Labs, afin de développer un système d'exploitation UNIX sur un DEC PDP-11. En 1978, Brian Kernighan et Dennis Richie publient la définition classique du C dans le livre The C Programming language [6]. Le C devenant de plus en plus populaire dans les années 80, plusieurs groupes mirent sur le marché des compilateurs comportant des extensions particulières. En 1983, l'ANSI (American National Standards Institute) décida de normaliser le langage ; ce travail s'acheva en 1989 par la définition de la norme ANSI C. Celle-ci fut reprise telle quelle par l'ISO (International Standards Organization) en 1990. C'est ce standard, ANSI C, qui est décrit dans le présent document.

En 1983 un groupe de développeurs de AT&T sous la direction de Bjarne Stroustrup a créé le langage C++. Le but était de développer un langage qui garderait les avantages de ANSI-C (portabilité, efficience) et qui permettrait en plus la programmation orientée objet. Depuis 1990 il existe une ébauche pour un standard ANSI-C++. Entre-temps AT&T a développé deux compilateurs C++ qui respectent les nouvelles déterminations de ANSI et qui sont considérés comme des quasi-standards (AT&T-C++ Version 2.1 [1990] et AT&T-C++ Version 3.0 [1992]).

Avantages

Le grand succès du langage C s'explique par les avantages suivants ; C est un langage :

1. universel : C n'est pas orienté vers un domaine d'applications spéciales, comme par exemple FORTRAN (applications scientifiques et techniques) ou COBOL (applications commerciales ou traitant de grandes quantités de données).
2. compact : C est basé sur un noyau de fonctions et d'opérateurs limité, qui permet la formulation d'expressions simples, mais efficaces.
3. moderne : C est un langage structuré, déclaratif et récursif ; il offre des structures de contrôle et de déclaration comparables à celles des autres grands langages de ce temps (FORTRAN, ALGOL68, PASCAL).
4. près de la machine : comme C a été développé en premier lieu pour programmer le système d'exploitation UNIX, il offre des opérateurs qui sont très proches de ceux du langage machine et des fonctions qui permettent un accès simple et direct aux fonctions internes de l'ordinateur (p.ex : la gestion de la mémoire).
5. rapide : comme C permet d'utiliser des expressions et des opérateurs qui sont très proches du langage machine, il est possible de développer des programmes efficients et rapides.
6. indépendant de la machine : bien que C soit un langage près de la machine, il peut être utilisé sur n'importe quel système en possession d'un compilateur C. Au début C était surtout le langage des systèmes travaillant sous UNIX, aujourd'hui C est devenu le langage de programmation standard dans le domaine des micro-ordinateurs.
7. portable : en respectant le standard ANSI-C, il est possible d'utiliser le même programme sur tout autre système (autre hardware, autre système d'exploitation), simplement en le recompilant.
8. extensible : C ne se compose pas seulement des fonctions standard ; le langage est animé par des bibliothèques de fonctions privées ou livrées par de nombreuses maisons de développement.

Désavantages

En C, nous avons la possibilité d'utiliser des expressions compactes et efficientes. D'autre part, nos programmes doivent rester compréhensibles pour nous-mêmes et pour d'autres. Ces deux exigences peuvent se contredire réciproquement.

De plus c'est un langage ancien qui a conservé la trace d'instructions désuètes et dangereuses.

1.4 Le développement de logiciel

Méthode.

1. Spécification du cahier des charges
2. Analyse du problème
3. Conception de l'approche, puis de l'algorithme de résolution
4. Ecriture de l'algorithme et implantation (et documentation)
5. Test
6. Maintenance et mises à jour

—— Exemple ——

Étude de cas : Conversion de miles en km.
[HK], ch.1, p.25-27

2 Premier contact avec le langage C

2.1 Un exemple de programme

—— Exemple ——

Un exemple de programme C : Exemple [HK] : ch.1, p.35

Un programme C comprend deux parties : des *directives de préprocesseur* et la *fonction principale.*

2.2 Directives du préprocesseur

Les directives du préprocesseur sont des commandes qui donnent des instructions au préprocesseur dont la tâche consiste à modifier le texte du programme C *avant* qu'il ne soit compilé.

Une directive de préprocesseur commence avec symbole (`#`) comme premier caractère non blanc. Elle n'est pas terminée par un point virgule.

Pour écrire une directive sur plusieurs lignes, il faut terminer chaque ligne qui n'est pas la dernière par le caractère \.

Les deux commandes les plus fréquentes sont `#include` et `#define`.

1. La directive `#include` procure au programme l'accès à des librairies de fonctions prédéfinies ou à des fichiers. La directive conduit le préprocesseur à insérer des définitions provenant de l'en-tête du fichier dans le programme avant l'étape de compilation.
   Par exemple, la directive :
   ```
   #include <stdio.h>            /* printf, scanf definitions */
   ```
   indique au préprocesseur que des noms utilisés dans le programme (comme `printf` ou `scanf`) se trouvent définis dans le fichier header standard `<stdio.h>`.
   La directive :
   ```
   #include "NomFichier"
   ```
   indique au préprocesseur d'inclure le contenu du fichier `NomFichier` dans le programme. Le fichier `NomFichier` est cherché dans le répertoire courant.
2. La directive `#define` associe des identifieurs avec des valeurs.
   Seules les valeurs qui ne changent pas devraient être définies par `#define`, car un programme C ne peut changer ces valeurs en cours d'exécution.
   - Les *macro substitutions* sont des macros sans argument, qui permettent de définir des constantes.
     `#define NomMacro ValeurDeSubstitution`
     Le préprocesseur remplace, dans le programme, chaque nom de macro `NomMacro` par la chaîne de caractères `ValeurDeSubstitution` qui lui est associée.
     Exemple :
     `#define Taille_Maximale 500`

   - La définition de *macro-fonctions* ressemble à la définition de symboles, mais elle fait intervenir la notion de paramètres.
     Par exemple, la directive :
     `#define carre(a) a*a`
     induira le préprocesseur à remplacer dans la suite tous les textes de la forme :
     `carre(x)`
     dans lesquels `x` représente en fait un symbole quelconque par :
     `x * x`

     La macro précédente ne disposait que d'un seul paramètre, mais il est possible d'en faire intervenir plusieurs en les séparant par des virgules. Par exemple, avec :
     `#define dif(a,b) a-b`
     Nous aurons les transformations suivantes :

`dif(x,z)`	deviendra	`x-z`
`dif(valeur+9,n)`	deviendra	`valeur+9-n`

     On peut imbriquer les définitions comme dans :
     `dif(carre(p),carre(q))`
     qui sera d'abord remplacé par :
     `dif(p*q,p*q)`

puis par :
`p*q-p*q`

Néanmoins, malgré la puissance de tte directive, il ne faut pas oublier qu'il ne s'agit de substitution de texte.

Les **commentaires**.

Entre `/*` et `*/`.

2.3 La fonction main

[HK], ch.2, p.37-

Les deux lignes d'en-tête :

```
int
main(void)
```

marquent le début de la fonction principale où l'exécution du programme commence. Tout programme C possède une fonction principale. Les lignes suivantes forment le *corps* de la fonction et se trouvent entre parenthèses {, }.

Un **corps de fonction** possède deux parties : des déclarations et des instructions exécutables.

1. Les **déclarations** disent au compilateur quelles sont les cellules mémoires à réserver pour la fonction (par exemple `miles` et `kms` dans l'exemple vu plus haut).
2. Les **instructions exécutables** sont traduites en langage machine avant d'être exécutées par l'ordinateur.

La fonction principale contient des symboles de *ponctuation* et des *symboles spéciaux* (`*, =`). Les *virgules* séparent les éléments d'une liste, les *points-virgules* séparent les instructions et les *parenthèses* marquent le début et la fin du corps de la fonction `main`.

EXEMPLE

Hello world

2.4 Identificateurs

Mots réservés

Les mots réservés sont des identificateurs des librairies standard, et de places mémoire spéciales. Ils apparaissent tous en minuscules et ont des significations spéciales pour C. Ils ne peuvent être utilisés pour des fins différentes.

Exemples :

Mots réservés	Signification
`int`	entier ; indique que la fonction principale retourne une valeur entière
`void`	indique que la fonction principale ne reçoit pas de données du système d'exploitation
`double`	indique que les cellules mémoire stockent des nombres réels
`return`	retourne le contrôle depuis la fonction principale au système d'exploitation

Identifieurs standard

Les identificateurs standard sont des mots qui ont une signification spéciale en C (comme `scanf` ou `printf`) mais qui peuvent cependant être redéfinis. `NON RECOMMANDÉ ! !`.

Identificateurs définis par l'utilisateur

Les identificateurs servent à désigner les différents "objets" manipulés par le programme : variables, fonctions, etc. 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 distinguées.

La norme ANSI prévoit qu'au moins les 31 premiers caractères des identificateurs sont "significatifs", c'est-à-dire permettent de distinguer deux identificateurs.

Exercices

[HK], ch.2, p.39.

Identificateurs invalides. Donnez-en la raison.

Remarque : **Bonne conduite en programmation**

Choisir des identificateurs faciles à lire et interpréter.

Exemples :

`salaire`

`Euros_par_heure` plutôt que `eurosparheure`

2.5 Exercices

[HK] : ch.2 ; p.41

3 Structure d'un programme C

3.1 Déclaration de variables et types de données

3.1.1 Déclaration de variables

[HK] : ch.2 ; p.41

Les cellules mémoire utilisées pour stocker les entrée d'un programme et les sorties produites par son exécution sont appelées *variables* car les valeurs stockées peuvent changer durant l'exécution du programme.

Les déclarations de variable dans un programme C communiquent au compilateur les noms des variables utilisées dans le programme. Elles indiquent aussi quel type d'information va être stocké et comment cette information sera représentée en mémoire.

Par exemple, les déclarations de variable :

```
double  miles ;       /* input - distance in miles.           */
double  kms ;         /* output - distance in kilometres.     */
```

procurent les noms de deux variables (`miles` et `kms`) utilisées pour stocker des nombres réels.

Une déclaration de variable commence par un identificateur de type (par exemple `double`) qui dit au compilateur le type de données stocké dans cette cellule.

Autres exemples :

```
int     compteur,X,Y ;
float   hauteur,largeur ;
double  masse_atomique ;
char    touche ;
int     t_pressee ;
```

Toutes les variables utilisées dans un programme doivent être déclarées.

Voir [HK], ch.2, p.41

Exercices

[HK], ch.2, p.41.

Lesquels des identificateurs suivants sont :

- (a) des mots réservés de C
- (b) des identificateurs standard
- (c) utilisés conventionnellement comme des noms de constantes macro
- (d) valides
- (e) invalides

```
void              MAX_ENTRIES  double  G       Sue's
return            printf       xyz123  part#2  "char"  #insert
this_is_a_long_one
```

Réponse :

- (a) : `void, double, return`
- (b) : `printf`
- (c) : `MAX_ENTRIES, G`
- (d) : `time, xyz123, this_is_a_long_one`
- (e) : `Sue's, part#2, "char", #include`

3.1.2 Initialisation des variables

En C, il est possible d'initialiser les variables lors de leur déclaration :

```
int    MAX = 1023 ;
char   TAB = '\t' ;
float  X = 1.05e-4 ;
```

En utilisant l'attribut `const`, nous pouvons indiquer que la valeur d'une variable ne change pas au cours d'un programme :

```
int     MAX = 767 ;
double  e = 2.71828182845905 ;
char    NEWLINE = '\n' ;
```

3.1.3 Initialisation des variables comme constantes

Le mot clé `const` permet de déclarer que la valeur d'une variable ne doit pas être modifiée pendant l'exécution d'un programme.

Une variable déclarée avec le mot clé `const` doit être initialisée dès sa définition.

Exemple

```
#include <stdio.h>
int main()
{
   const int n=10 ;
   n=5 ; //erreur de compilation : n a été déclaré const
   int p=2*n ;
   return 0 ;
}
```

3.2 Les types prédéfinis

[HK] : ch.2 ; p.42

Un type de données définit à la fois un ensemble de valeurs et un ensemble d'opérations possibles sur ces valeurs. Un type de données standard en C est un type prédéfini, comme `char`, `int` ou `double`. Les types `int` et `double` sont utilisés comme des abstractions pour les nombres entiers et pour les nombres réels.

Attention. Les constantes numériques en C sont considérées comme des nombres positifs. Quoiqu'il soit possible d'écrire une expression telle que `-10500`, C considère le signe `-` comme un opérateur de négation et non comme le signe -.

3.2.1 Le type caractère

Les caractères sont de type `char`.

Les caractères sont représentés en mémoire sur 8 bits :
- domaine de valeurs du type char de -128 à 127 ;
- domaine de valeurs du type unsigned char de 0 à 255.

Les valeurs de type char sont traitées comme des entiers et peuvent être utilisées dans des expressions entières.

3.2.2 Les types entiers

À cause de la taille finie des cellules mémoire, tous les entiers ne peuvent pas être représentés. La norme ANSI stipule que le type `int` doit inclure au moins les valeurs allant de `-32767` à `32767`. Les opérations arithmétiques classiques (addition, soustraction, multiplication, division) et la comparaison de deux entiers sont prédéfinies pour ce type.

- **Les types entiers signé**.

Type	**Taille (bits)**	**domaine de valeurs** -2^{n-1} à $2^{n-1}-1$
`short int`	16 bits	-32768 à 32767
`int`	16 ou 32 bits	
`long int`	32 bits	-2 147 483 648 à 2 147 483 647

Le choix d'un des trois types entier `short int`, `int` ou `long int` peut être guidé par les trois considérations suivantes :
– L'intervalle de valeurs nécessaires,
– La quantité d'espace mémoire disponible,
– La vitesse du programme.
Le type `int` est traditionnement le type entier "standard" du mangage C et les opérations sur ce type sont généralement les plus efficaces. Cependant la représentation en mémoire des entiers de type `int` dépend du compilateur C utilisé, ce qui pose des problèmes de portabilité du programme C.
E.g. Le type `int` est représenté sur 16 bits avec le compilateur de Dev C++.

- **les types entier non signé**

Type	**Taille (bits)**	**domaine de valeurs** -2^{n-1} à $2^{n-1}-1$
`unsigned short int`	16 bits	0 à 65 535
`unsigned int`	16 ou 32 bits	
`unsigned long int`	32 bits	0 à 4 294 967 295

3.2.3 Les types flottants

Il existe deux types réels standard : `float` et `double`. Certains compilateurs proposent aussi le type réel `long double`.

Type	**Taille (bits)**	**domaine de valeurs** (pour les valeurs positives)
`float`	32 bits	3.4 E^{-38} à 3.4 E^{38}
`double`	64 bits	1.7 E^{-308} à 1.7 E^{308}
`long double`	64 ou 80 bits	

Exercices

[HK], ch.2, p.43

Identifier les constantes valides et invalides.

3.3 Les constantes

3.3.1 Les constantes entières

3.3.2 Les constantes réelles

3.3.3 Les constantes caractères

3.3.4 Les constantes chaînes de caractères

Exercices

[HK] : ch.2 ; p.44

1- Écrire en notation décimale : `103e-4`, `1.23445e+6`, `123.45e+3`

2- Écrire en notation scientifique : `1300`, `123.45`, `0.00426`

3- Quels seraient les meilleurs types de donnée pour

- le calcul de l'aire d'un cercle
- le nombre de voitures passant dans un tunnel par heure
- les premières lettres du nom de famille

3.4 Les opérateurs

3.4.1 L'affectation

3.4.2 Les opérateurs arithmétiques

[HK] : ch.2 ; p.58-70

Utilisées pour manipuler des données de type `int` ou `double`.

3.4.2.1 Principaux opérateurs

Opérateurs arithmétiques	**Signification**	**Exemples**
`+`	addition	`5 + 2` est `7` `5.0 + 2.0` est `7.0`
`-`	soustraction	`5 - 2` est `3` `5.0 - 2.0` est `3.0`
`*`	multiplication	`5 * 2` est `10` `5.0 * 2.0` est `10.0`
`/`	division	`5 / 2` est `2` `5.0 / 2.0` est `2.5`
`%`	reste	`5 % 2` est `2`

3.4.2.2 Les opérateurs / et %

Attention quand les opérateurs `/` et `%` sont utilisés avec des opérande à la fois positif et négatif. Le résultat dépend d'une implantation à l'autre!!

Par exemple, voici ce que donne la division entière pour quelques exemples :

`3 / 15 = 0`	`18 / 3 = 6`
`15 / 3 = 5`	`16 / -3` varie
`16 / 3 = 5`	`0 / 4 = 0`
`17 / 3 = 5`	`4 / 0` indéfini

Par exemple, voici ce que donne le reste pour quelques exemples :

`3 % 5 = 3`	`5 % 3 = 2`
`4 % 5 = 4`	`5 % 4 = 1`
`5 % 5 = 0`	`15 % 5 = 0`
`6 % 5 = 1`	`15 % 6 = 3`
`7 % 5 = 2`	`15 % -7` varie
`8 % 5 = 3`	`15 % 0` est indéfini

3.4.2.3 Type de donnée d'une expression

Le type de donnée associé à une variable doit être spécifié dans une déclaration. Mais comment C détermine-t-il le type du résultat d'une expression ?

Celui-ci dépend des types des opérandes.

E.g. `Nb_mois + Nb_pommes` sera de type `int` si les deux variables présentes sont chacune de type `int`.

Les opérations avec types mixtes

Quand une affectation est réalisée, l'expression à la droite du symbole (`=`) est d'abord évaluée, puis le résultat est affecté à la variable figurant à gauche de `=`.

Règles de conversion automatique

Conversions automatiques lors d'une opération avec,

1. deux entiers : D'abord, les types `char` et `short` sont convertis en `int`. Ensuite, l'ordinateur choisit le plus large des deux types dans l'échelle suivante : `int`, `unsigned int`, `long`, `unsigned long`
2. un entier et un rationnel : Le type entier est converti dans le type du rationnel.
3. deux rationnels : L'ordinateur choisit le plus large des deux types selon l'échelle suivante : `float`, `double`, `long double`
4. affectations et opérateurs d'affectation : Lors d'une affectation, le résultat est toujours converti dans le type de la destination. Si ce type est plus faible, il peut y avoir une perte de précision.

Exemple

Observons les conversions nécessaires lors d'une simple division : `int X ;`

```
float A=12.48 ;
char B=4 ;
X=A/B ;
```

`B` est converti en `float` (règle 2). Le résultat de la division est du type `float` (valeur 3.12) et sera converti en `int` avant d'être affecté à `X` (règle 4), ce qui conduit au résultat `X=3` .

Phénomènes imprévus ...

Le mélange de différents types numériques dans un calcul peut inciter à ne pas tenir compte des phénomènes de conversion et conduit parfois à des résultats imprévus ...

Exemple :

Dans cet exemple, nous divisons 3 par 4 à trois reprises et nous observons que le résultat ne dépend pas seulement du type de la destination, mais aussi du type des opérandes.

```
 char A=3 ;
int B=4 ;
float C=4 ;
float D,E ;
char F ;
D = A/C ;
E = A/B ;
F = A/C ;
```

- Pour le calcul de `D, A` est converti en `float` (règle 2) et divisé par `C`. Le résultat (0.75) est affecté à `D` qui est aussi du type `float`. On obtient donc : `D=0.75`
- Pour le calcul de `E, A` est converti en `int` (règle 1) et divisé par `B`. Le résultat de la division (type `int`, valeur 0) est converti en `float` (règle 4). On obtient donc : `E=0.000`
- Pour le calcul de `F, A` est converti en `float` (règle 2) et divisé par `C`. Le résultat (0.75) est retraduit en `char` (règle 4). On obtient donc : `F=0`

Perte de précision

Lorsque nous convertissons une valeur en un type qui n'est pas assez précis ou pas assez grand, la valeur est coupée sans arrondir et sans nous avertir ...

Exemple :

```
unsigned int A = 70000 ;
/* la valeur de A sera : 70000 mod 65536 = 4464 */
```

Exemple. [HK], ch.2, p.61-62

Exercices

[C. Delannoy, "exos"], ex-2, p.4

[C. Delannoy, "exos"], ex-3, p.5

3.4.2.4 La conversion de type.

[HK] : ch.2 ; p.62-63

Le langage C permet au programmeur de changer le type d'une expression en spécifiant le type désiré entre parenthèses devant l'expression.

Ainsi, `(int x) (x + 0.5) ;` s'évaluera à l'entier le plus proche de `x`.

Il est possible de convertir explicitement une valeur en un type quelconque en forçant la transformation à l'aide de la syntaxe (*Casting* (conversion de type forcée))

```
(<Type>) <Expression>
```

Exemple

Nous divisons deux variables du type entier. Pour avoir plus de précision, nous voulons avoir un résultat de type rationnel. Pour ce faire, nous convertissons l'une des deux opérandes en `float`. Automatiquement C convertira l'autre opérande en `float` et effectuera une division rationnelle :

```
 char A=3 ;
int B=4 ;
float C ;
C = (float)A/B ;
```

La valeur de `A` est explicitement convertie en `float`. La valeur de `B` est automatiquement convertie en `float` (règle 2). Le résultat de la division (type rationnel, valeur 0.75) est affecté à `C`.

Résultat : `C=0.75`

Attention !

Les contenus de `A` et de `B` restent inchangés ; seulement les valeurs utilisées dans les calculs sont converties !

3.4.3 Les opérateurs logiques booléens

Les résultats des opérations de comparaison et des opérateurs logiques sont du type int :

- la valeur 1 correspond à la valeur booléenne vrai
- la valeur 0 correspond à la valeur booléenne faux

Les opérateurs logiques considèrent toute valeur différente de zéro comme vrai et zéro comme faux :

`&&`	et logique (and)
`||`	ou logique (or)
`!`	négation logique (not)

Exemple

```
32 && 2.3        → 1
 !65.34          → 0
0 || !(32 > 12)  → 0
```

3.4.4 Les opérateurs logiques bit à bit

3.4.5 Les opérateurs de comparaison

`==`	égal à ?
`!=`	différent de ?
`<, <=, >, >=`	plus petit que ? ...

3.4.6 Les opérateurs d'affectation composée

3.4.7 Les opérateurs d'incrémentation et de décrémentation

3.4.8 L'opérateur virgule

3.4.9 L'opérateur conditionnel ternaire

3.4.10 L'opérateur de conversion de type

3.4.11 L'opérateur adresse

3.4.12 Règles de priorité des opérateurs

Règles de priorité d'évaluation.

Priorité 1 (la plus forte)	`()`
Priorité 2	`! ++ --`
Priorité 3	`* / %`
Priorité 4	`+ -`
Priorité 5	`< <= > >=`
Priorité 6	`== !=`
Priorité 7	`&&`
Priorité 8	`||`
Priorité 9 (la plus faible) :	`= += -= *= /= %= \\`

[HK], ch.2, p.64-67 pour des exemples.

3.5 Instructions

[HK] : ch.2 ; p.45-55

Une instruction en C est :

- une **instruction simple** toujours terminée par un point-virgule et pouvant être librement écrite sur plusieurs lignes ;
- une **instruction composée**, encore appelée bloc d'instructions, qui est une suite d'instructions placées entre accolades.

La *portée d'une variable* dépend de l'endroit où la variable est définie (voir plus bas en section 3.5.2 et suivantes).

3.5.1 Affectation de variables

En C, le symbole `=` est l'opérateur d'affectation (donc de signification radicalement différente de la signification en math).

L'affectation avec des valeurs constantes

```
LONG = 141 ;
PI = 3.1415926 ;
NATION = 'L' ;
```

L'affectation avec des valeurs de variables

```
VALEUR = X1A ;
LETTRE = COURRIER ;
```

L'affectation avec des valeurs d'expressions

```
AIRE = PI*pow(R,2) ;
MOYENNE = (A+B)/2 ;
UN = pow(sin(X),2) + pow(cos(X),2) ;
RES = 45+5*X ;
PLUSGRAND = (X>Y) ;
CORRECT = ('a' == 'a') ;
```

Remarque : le test d'égalité en C se formule avec deux signes d'égalité == , l'affectation avec un seul = .

L'affectation avec des expressions spéciales

En pratique, nous retrouvons souvent des affectations comme :

```
i = i + 2 ;
```

En C, nous utiliserons plutôt la formulation plus compacte :

```
 i += 2 ;
```

L'opérateur `+=` est un opérateur d'affectation.

Pour la plupart des expressions de la forme :

`expr1 = (expr1) op (expr2)`

il existe une formulation équivalente qui utilise un opérateur d'affectation :

`expr1 op= expr2`

Table d'opérateurs d'affectation :

`+=`	ajouter à
`-=`	diminuer de
`*=`	multiplier par
`%=`	modulo

Opérateurs d'incrémentation et de décrémentation

Les affectations les plus fréquentes sont du type :

`I = I + 1` et `I = I - 1`

En C, nous disposons de deux opérateurs inhabituels pour ces affectations :

`I++` ou `++I` pour l'incrémentation (augmentation d'une unité)
`I--` ou `--I` pour la décrémentation (diminution d'une unité)

Les opérateurs `++` et `--` sont employés dans les cas suivants :

- incrémenter/décrémenter une variable (p.ex : dans une boucle). Dans ce cas il n'y a pas de différence entre la notation préfixe (++I –I) et la notation postfixe (I++ I–).
- incrémenter/décrémenter une variable et en même temps affecter sa valeur à une autre variable. Dans ce cas, nous devons choisir entre la notation préfixe et postfixe :

  `X = I++` passe d'abord la valeur de I à X et **incrémente après**
  `X = I--` passe d'abord la valeur de I à X et **décrémente après**
  `X = ++I` **incrémente d'abord** et passe la valeur incrémentée à X
  `X = --I` **décrémente d'abord** et passe la valeur décrémentée à X

Exemple :

Supposons que la valeur de `N` est égal à 5 :

Incrém. postfixe : `X = N++ ;` Résultat : `N=6` et `X=5`

Incrém. préfixe : `X = ++N ;` Résultat : `N=6` et `X=6`

3.5.2 Les variables automatiques

Une variable définie dans un bloc d'instructions est une **variable locale temporaire**. En C, une variable locale temporaire a l'attribut `auto` (pour automatique) et est appelée **variable automatique**.

La portée d'une variable automatique va de la ligne de sa définition jusqu'à la fin du bloc d'instructions dans lequel elle est définie. Plus précisément, lors de l'exécution, une variable automatique est créée à l'entrée du bloc d'instructions dans lequel elle est définie et est détruite à la sortie de ce bloc.

Une variable automatique n'est jamais initialisée par défaut.

Exemple

```
#include <stdio.h>
int main()
{
    int n=10,m=20 ;
    int n=5 ;  //erreur de compilation : redéfinition de n
    {         float n=3.5 ; int m=3 ;  //redéfinition de n et de m : A EVITER
        int p=20 ;
        printf("n=%f", n) ; //affiche 3.500000
        printf("m=%d", m) ;  //affiche 3
        printf("p=%d", p) ;  //affiche 20
    }
    printf("n=%f", n) ;  //affiche 10
```

```
    printf("m=%d", m) ; //affiche 20
    printf("p=%d", p) ;  //erreur de compilation : p inconnu
}
```

3.5.3 Les variables externes

Une variable définie à l'extérieur de tout bloc d'instructions (y compris la fonction `main`) est une **variable globale**.

La portée d'une variable globale définie dans un fichier s'étend de l'endroit où elle est définie jusqu'à la fin du fichier. Plus précisément, une variable globale est créée au début de l'exécution du programme et est détruite à la fin du programme.

Une variable globale est initialisée par défaut à zéro.

La portée d'une variable globale peut être étendue à tout fichier dans lequel la variable est déclarée avec l'attribut extern. De ce fait, une variable globale en C est appelée **variable externe**.

Il est important de **distinguer la déclaration d'une variable de sa définition**. Comme nous l'avons déjà dit, la définition d'une variable correspond à la réservation de l'emplacement mémoire nécessaire à la représentation de la variable. La déclaration externe de cette variable ne la redéfinit pas : elle indique seulement les caractéristiques de la variable nécessaires à son utilisation. Une variable ne peut être déclarée avec l'attribut `extern` que s'il existe, par ailleurs, une définition de cette variable.

Exemple

Fichier A

```
int n=0;
float x=0;

void Somme()
{
    ...
    /*n et x sont accessibles*/
}
```

Fichier B

```
extern int n;

int main()
{
    ...
    /*n est accessible*/
}
```

Fig. 1.1: Exemple.

3.5.4 Les variables statiques

Une **variable statique** est une variable déclarée avec l'attribut `static`.

Une variable statique est créée une seule fois au début de l'exécution du programme et est détruite à la fin du programme. S'il existe une ou des valeur(s) d'initialisation pour la variable, celle-ci n'est initialisée qu'une seule fois au début de l'exécution du programme et la variable conserve sa valeur. La mémorisation d'une variable statique au cours de l'exécution du programme est permanente.

Une variable statique peut être interne ou externe. La portée d'une variable statique interne est limitée au bloc d'instructions dans lequel elle est définie. La portée d'une variable statique externe est limitée au fichier dans lequel elle est définie.

Une variable statique est initialisée par défaut à zéro.

Exemple

```
#include <stdio.h>
int main() {
     int n=5, total=0 ;
    while(1) {
```

```
        static int i=1 ;  //variable statique initialisée une seule fois
        total=total+i ;
        if(i==n) break ;
        i=i+1 ;}
    printf("La somme des entiers de 1 a %d est %d\n",n,total) ;
}
```

Le résultat de l'exécution de ce programme dans l'environnement de développement Dev-C++ est :
La somme des entiers de 1 a 5 est 15
Exemple :

FIG. 1.2: Exemple.

La portée de la variable statique `x`, définie dans le fichier A, est limitée au fichier A.

3.5.5 L'instruction return

La dernière ligne dans l'exemple de programme étudié ici est :

```
return (0) ;
```

Elle transfère le contrôle du programme au système d'exploitation. La valeur entre parenthèses, ici `0`, est considérée comme le résultat de l'exécution du programme principal `main`, et il indique que le programme s'est déroulé sans erreur.

4 Les instructions de branchement conditionnel

On appelle instruction de contrôle toute instruction qui permet de contrôler le fonctionnement d'un programme. Parmi les instructions de contrôle, on distingue les *instructions de branchement* et les *boucles*. Les instructions de branchement permettent de déterminer quelles instructions seront exécutées et dans quel ordre.

4.1 Branchement conditionnel if---else

La forme la plus générale est celle-ci :

```
if (expression-1 )
    instruction-1
else if (expression-2 )
    instruction-2
    ...
else if (expression-n )
    instruction-n
else
    instruction-Y
```

avec un nombre quelconque de `else if ( ... )`. Le dernier `else` est toujours facultatif. La forme la plus simple est

```
if (expression )
    instruction
```

Chaque instruction peut être un bloc d'instructions.

4.2 Branchement multiple switch

Sa forme la plus générale est celle-ci :

```
switch (expression )
    {case constante-1 :
       liste d'instructions 1
       break ;
    case constante-2 :
       liste d'instructions 2
       break ;
    ...
    case constante-n :
       liste d'instructions n
       break ;
    default :
       liste d'instructions Y
       break ;
    }
```

Si la valeur de `expression` est égale à l'une des `constantes`, la `liste d'instructions` correspondant est exécutée. Sinon la `liste d'instructions Y` correspondant à `default` est exécutée. L'instruction `default` est facultative.

5 Les boucles

Les *boucles* permettent de répéter une série d'instructions tant qu'une certaine condition n'est pas vérifiée.

5.1 Boucle while

La syntaxe de `while` est la suivante :

```
while (expression )
    instruction
```

Tant que `expression` est vérifiée (i.e., non nulle), instruction est exécutée. Si `expression` est nulle au départ, `instruction` ne sera jamais exécutée. `instruction` peut évidemment être une instruction composée. Par exemple, le programme suivant imprime les entiers de 1 à 9.

Exemple

```
 i = 1 ;
 while (i < 10)
     {
     printf("\n i = %d",i) ;
     i++ ;
     }
```

5.2 Boucle do---while

Il peut arriver que l'on ne veuille effectuer le test de continuation qu'après avoir exécuté l'instruction. Dans ce cas, on utilise la boucle `do---while`. Sa syntaxe est

```
do
```

```
    instruction
while (expression ) ;
```

Ici, `instruction` sera exécutée tant que `expression` est non nulle. Cela signifie donc que `instruction` est toujours exécutée au moins une fois. Par exemple, pour saisir au clavier un entier entre 1 et 10 :

Exemple

```
int a ;

do
    {
    printf("\n Entrez un entier entre 1 et 10 : ") ;
    scanf("%d",&a) ;
    }
while ((a <= 0) || (a > 10)) ;
```

5.3 Boucle for

La syntaxe de `for` est :

```
for (expr 1 ;expr 2 ;expr 3)
instruction
```

Une version équivalente plus intuitive est :

```
expr 1 ;
while (expr 2 )
    {instruction
    expr 3 ;
    }
```

Par exemple, pour imprimer tous les entiers de 0 à 9, on écrit :

```
for (i = 0 ; i < 10 ; i++)
    printf("\n i = %d",i) ;
```

A la fin de cette boucle, `i` vaudra `10`.

Les trois expressions utilisées dans une boucle `for` peuvent être constituées de plusieurs expressions séparées par des virgules. Cela permet par exemple de faire plusieurs initialisations à la fois.

Par exemple, pour calculer la factorielle d'un entier, on peut écrire :

Exemple

```
int n, i, fact ;
for (i = 1, fact = 1 ; i <= n ; i++)
    fact *= i ;
printf("%d ! = %d \n",n,fact) ;
```

On peut également insérer l'instruction `fact *= i ;` dans la boucle `for` ce qui donne :

Exemple

```
int n, i, fact ;
for (i = 1, fact = 1 ; i <= n ; fact *= i, i++) ;
   printf("%d ! = %d \n",n,fact) ;
```

On évitera toutefois ce type d'acrobaties qui n'apportent rien et rendent le programme difficilement lisible.

6 Les instructions de branchement non conditionnel

6.1 Branchement non conditionnel `break`

6.2 Branchement non conditionnel `continue`

6.3 Branchement non conditionnel `goto`

7 Les fonctions d'entrées-sorties classiques

Les données peuvent être affectées à des cellules mémoire selon deux modes :
- par affectation de variable,
- par copie de données obtenues à travers un dispositif d'entrée, en utilisant une fonction telle que `scanf`.

Les transferts de données depuis un dispositif d'entrée sont appelées *opérations d'entrée*. Les données résultant de l'exécution d'un programme peuvent être rendues disponibles sur un dispositif de sortie grâce à des *opérations de sortie*.

En C, toutes les opérations d'entrée-sortie sont réalisées par des fonctions spéciales appelées *fonctions d'entrée-sortie*. Les fonctions standard sont fournies par la bibliothèque standard d'entrée-sortie accessible par la directive :

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

Cette section présente la fonction d'entrée `scanf` et la fonction de sortie `printf`.

7.1 La fonction d'écriture `printf`

[HK] ch.2, p. 48-51

Le format général de la fonction `printf` est le suivant :

```
printf(format, val1, val2, ..., valn)
```

- `val1, val2, ..., valn` représentent les valeurs à afficher ;
- format est une chaîne de caractères contenant les codes de mise en forme associés, dans l'ordre, aux arguments vali de la fonction `printf`.

Un code de mise en forme pour une valeur donnée précise le format d'affichage de la valeur. Il est précédé du caractère % :

- `%d` permet d'afficher une valeur entière,
- `%f` permet d'afficher une valeur réelle,
- `%c` permet d'afficher un caractère,
- `%s` permet d'afficher une chaîne de caractères,
- `%%` permet d'afficher le caractère «

7.1.0.1 Les séquences d'échappement L'impression et l'affichage de texte peuvent être contrôlés à l'aide de séquences d'échappement. Une séquence d'échappement est un couple de symboles dont le premier est le signe d'échappement ’: Au moment de la compilation, chaque séquence d'échappement est traduite en un caractère de contrôle dans le code de la machine. Comme les séquences d'échappement sont identiques sur toutes les machines, elles nous permettent d'écrire des programmes portables, c.-à-d. : des programmes qui ont le même effet sur toutes les machines, indépendamment du code de caractères utilisé.

\a	sonnerie	\\	trait oblique
\b	curseur arrière	\?	point d'interrogation
\t	tabulation	\'	apostrophe
\n	nouvelle ligne	\"	guillemets
\r	retour en début de ligne	\f	saut de page (imprimante)
\0	NUL	\v	tabulateur vertical

Le caractère NUL

La constante \0 qui a la valeur numérique zéro a une signification spéciale dans le traitement et la mémorisation des chaînes de caractères : En C le caractère \0 définit la fin d'une chaîne de caractères.

Exercices

[HK] : ch.2 ; p.54

7.2 La fonction de saisie scanf

La fonction `scanf` permet de saisir des données au clavier et de les stocker aux adresses spécifiées par les arguments de la fonctions.

```
scanf("chaîne de contrôle",argument-1,...,argument-n)
```

La `chaîne de contrôle` indique le format dans lequel les données lues sont converties. Elle ne contient pas d'autres caractères (notamment pas de \n). Comme pour `printf`, les conversions de format sont spécifiées par un caractère précédé du signe %. Les formats valides pour la fonction `scanf` diffèrent légèrement de ceux de la fonction `printf`.

Les données à entrer au clavier doivent être séparées par des blancs ou des `<RETURN>` sauf s'il s'agit de caractères. On peut toutefois fixer le nombre de caractères de la donnée à lire. Par exemple `%3s` pour une chaîne de 3 caractères, `%10d` pour un entier qui s'étend sur 10 chiffres, signe inclus.

Exemple

```
#include <stdio.h>
main()
{  int i ;
printf("entrez un entier sous forme hexadecimale i = ") ;
scanf("%x",&i) ;
printf("i = %d\n",i) ;
}
```

Si on entre au clavier la valeur 1a, le programme affiche i = 26.

format	type d'objet pointé	représentation de la donnée saisie
%d	int	décimale signée
%hd	short int	décimale signée
%ld	long int	décimale signée
%u	unsigned int	décimale non signée
%hu	unsigned short int	décimale non signée
%lu	unsigned long int	décimale non signée
%o	int	octale
%ho	short int	octale
%lo	long int	octale
%x	int	hexadécimale
%hx	short int	hexadécimale
%lx	long int	hexadécimale
%f	float	flottante virgule fixe
%lf	double	flottante virgule fixe
%Lf	long double	flottante virgule fixe
%e	float	flottante notation exponentielle
%le	double	flottante notation exponentielle
%Le	long double	flottante notation exponentielle
%g	float	flottante virgule fixe ou notation exponentielle
%lg	double	flottante virgule fixe ou notation exponentielle
%Lg	long double	flottante virgule fixe ou notation exponentielle
%c	char	caractère
%s	char*	chaîne de caractères

7.3 Impression et lecture de caractères

Les fonctions `getchar` et `putchar` permettent respectivement de lire et d'imprimer des caractères. Il s'agit de fonctions d'entrées-sorties non formatées.

La fonction `getchar` retourne un `int` correspondant au caractère lu. Pour mettre le caractère lu dans une variable `caractere`, on écrit :

```
caractere = getchar() ;
```

Lorsqu'elle détecte la fin de fichier, elle retourne l'entier `EOF` (End Of File), valeur définie dans la librairie `stdio.h`. En général, la constante `EOF` vaut `-1`.

La fonction `putchar` écrit caractere sur la sortie standard :

```
putchar(caractere) ;
```

Elle retourne un `int` correspondant à l'entier lu ou à la constante `EOF` en cas d'erreur.

Par exemple, le programme suivant lit un fichier et le recopie caractère par caractère à l'écran.

— Exemple —

```
#include <stdio.h>
main()
{
    char c ;

    while ((c = getchar()) != EOF)
       putchar(c) ;
}
```

Pour l'exécuter, il suffit d'utiliser l'opérateur de redirection d'Unix :

```
programme-executable < nom-fichier
```

Notons que l'expression `(c = getchar())` dans le programme précédent a pour valeur la valeur de l'expression `getchar()` qui est de type `int`. Le test `(c = getchar()) != EOF` compare donc bien deux objets de type `int` (signés).

Ce n'est par contre pas le cas dans le programme suivant :

EXEMPLE

```
#include <stdio.h>
main()
{
    char c ;
    do
       {
       c = getchar() ;
       if (c != EOF)
          putchar(c) ;
       }
    while (c != EOF) ;
}
```

Ici, le test `c != EOF` compare un objet de type `char` et la constante `EOF` qui vaut `-1`. Si le type `char` est non signé par défaut, cette condition est donc toujours vérifiée. Si le type `char` est signé, alors le caractère de code `255`, y, sera converti en l'entier `-1`. La rencontre du caractère y sera donc interprétée comme une fin de fichier. Il est donc recommandé de déclarer de type `int` (et non `char`) une variable destinée à recevoir un caractère lu par `getchar` afin de permettre la détection de fin de fichier.

7.4 Le formatage des nombres en sortie

[HK] : ch.2 ; p.72-75

Exercices

[HK] : ch.2 ; p.75

[C. Delannoy, "exos"], ex-6, p.7

8 Quelques leçons générales

Il existe très peu de contraintes dans l'écriture d'un programme C. Toutefois ne prendre aucune précaution aboutirait à des programmes illisibles. Aussi existe-t-il un certain nombre de conventions.

- On n'écrit qu'une seule instruction par ligne : le point virgule d'une instruction ou d'une déclaration est toujours le dernier caractère de la ligne.
- Les instructions sont disposées de telle façon que la structure modulaire du programme soit mise en évidence. En particulier, une accolade ouvrante marquant le début d'un bloc doit être seule sur sa ligne ou placée à la fin d'une ligne. Une accolade fermante est toujours seule sur sa ligne.
- On laisse un blanc
  - entre les mots-clefs `if`, `while`, `do`, `switch` et la parenthèse ouvrante qui suit,
  - après une virgule,
  - de part et d'autre d'un opérateur binaire.
- On ne met pas de blanc entre un opérateur unaire et son opérande, ni entre les deux caractères d'un opérateur d'affectation composée.
- Les instructions doivent être indentées afin que toutes les instructions d'un même bloc soient alignées. Le mieux est d'utiliser le mode C d'Emacs.

8.1 Bonne conduite en programmation

[HK], ch.2, p.56-57

Les espaces

Les espaces supplémentaires ne sont pas considérés comme significatifs par le compilateur C. C'est une opportunité pour les utiliser afin de rendre les programmes plus lisibles, sans changer leur sémantique.

Les commentaires

8.2 Les charges d'un programmeur

Même un programmeur utilisant C ne doit pas connaître tous les détails des méthodes de codage et de calcul, il doit quand même être capable de :

- choisir un type numérique approprié à un problème donné ; c.-à-d. : trouver un optimum de précision, de temps de calcul et d'espace à réserver en mémoire
- prévoir le type résultant d'une opération entre différents types numériques ; c.-à-d. : connaître les transformations automatiques de type que C accomplit lors des calculs
- prévoir et optimiser la précision des résultats intermédiaires au cours d'un calcul complexe ; c.-à-d. : changer si nécessaire l'ordre des opérations ou forcer l'ordinateur à utiliser un type numérique mieux adapté

EXEMPLE

Supposons que la mantisse du type choisi ne comprenne que 6 positions décimales (ce qui est très réaliste pour le type float), alors voir figure 1.3.

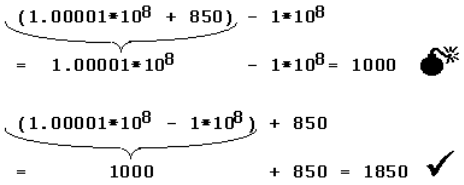


FIG. 1.3: Importance de l'ordre d'évaluation des opérations.

Exercices

[HK] : ch.2 ; p.67-70 : calcul de la valeur d'un tas de pièces

---

Exercices

[HK] : ch.2 ; p.71-72

8.3 Modes interactif et batch. Fichiers de données

8.4 Erreurs de programmation courantes

[HK] : ch.2 ; p.80-

8.4.1 Le debogage

8.4.2 Erreurs de syntaxe

8.4.3 Erreurs d'exécution

8.4.4 Erreurs non détectées

8.4.5 Erreurs de logique

8.4.6 Exercices

[HK] : ch.2 ; p.88-

9 Méthodologie de conception de programme

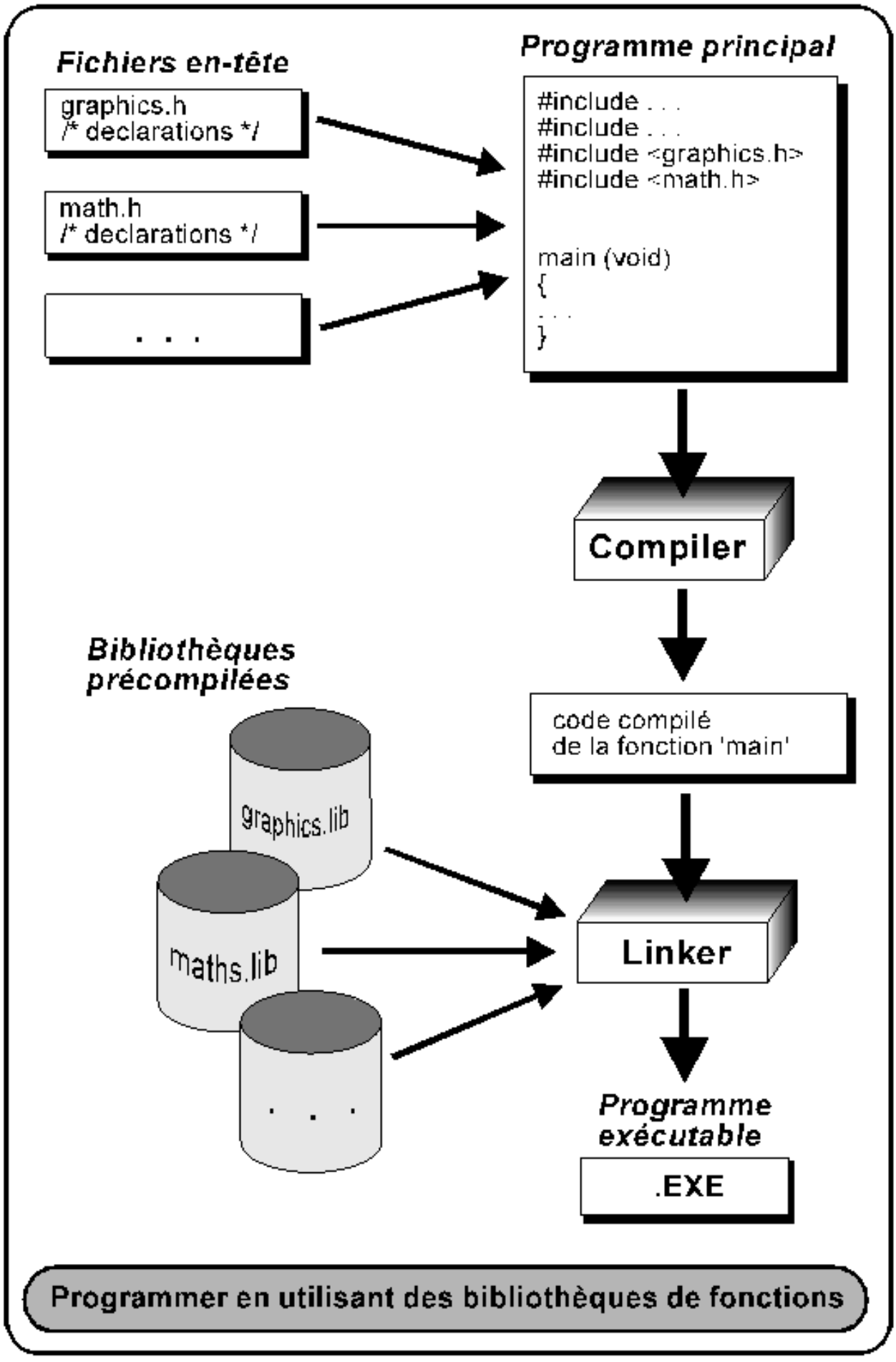
[HK] : ch.3 ; p.96-104

9.1 Compilation

[HK] : ch.1 ; p.19

9.2 Prise en main d'un environnement de programmation : Dev-C++

Voir la page : `http ://www.dil.univ-mrs.fr/ garreta/generique/autres/UtiliserDevCpp.html`


Fichiers en-tête
graphics.h
/* declarations */
math.h
/* declarations */
. . .
Programme principal
#include . . .
#include . . .
#include <graphics.h>
#include <math.h>
main (void)
{
. . .
}
Compiler
code compilé
de la fonction 'main'
Bibliothèques
précompilées
graphics.lib
maths.lib
. . .
Linker
Programme
exécutable
.EXE
Programmer en utilisant des bibliothèques de fonctions

TD1. Prise en main de l'environnement de développement Dev-C++

1 Compilation et exécution d'un programme sous Dos

Lancez l'invite de commande Dos et vérifiez que vous avez bien accès au compilateur C en tapant les commandes suivantes :

```
>cd C:\Dev-Cpp\bin\          //pour aller dans le répertoire C:\Dev-Cpp\bin\
C:\Dev-Cpp\bin>dir           //pour voir le contenu du répertoire
//Vérifiez que le fichier exécutable gcc.exe se trouve bien dans ce répertoire
C:\Dev-Cpp\bin>notepad       //pour ouvrir un éditeur de texte
```

Tapez le programme suivant dans l'éditeur de texte et le sauver sous N:\test.c.

```
#include <stdio.h>                    //bibliothèque d'entrées/sorties
int main() {
      printf("Hello World !\n"); return 0;
}
```

Nous allons à présent compiler et exécuter ce programme. Retournez sur l'invite de commande Dos et tapez les commandes suivantes :

```
C:\Dev-Cpp\bin>N:            //pour aller dans le répertoire N:\
N:\>dir                      //pour vérifier que test.c existe bien
//Compilez le programme à l'aide de la commande suivante :
N:\>C:\Dev-Cpp\bin\gcc.exe –c test.c –o test.o
N:\>dir                      //pour vérifier que test.o existe
//Générez l'exécutable du programme à l'aide de la commande suivante :
N:\>C:\Dev-Cpp\bin\gcc.exe test.o –o test.exe
N:\>dir                      //pour vérifier que test.exe existe
//Exécutez le programme à l'aide de la commande suivante
N:\>test.exe
Hello World !

N:\>test                     //autre moyen d'exécuter le programme
Hello World !

N:\>
```

Vous pouvez également utiliser les deux commandes suivantes au choix pour compiler et exécuter ce programme :

```
N:\>C:\Dev-Cpp\bin\gcc.exe test.c –o test.exe
N:\>test.exe
Hello World !
```

```
N:\>C:\Dev-Cpp\bin\gcc.exe test.c
N:\>dir                          //pour vérifier que a.exe existe
N:\>a.exe
Hello World !

N:\>
```

2 Préparation des fichiers et réglages

Dans votre espace disque personnel (N:), directement sous la racine N:\, copiez le répertoire MAP_C_eleves indiqué par l'enseignant :
ATTENTION, copiez le répertoire avec toute sa descendance directement sous N:\ de manière à avoir, dans votre espace disque, le répertoire N:\MAP_C_eleves.

Lancez l'environnement de développement "Dev-C++" et procédez aux réglages suivants :

- Options d'environnement (Menu : Outils – options d'environnement) :
  - onglet général : décochez la case « C++ par défaut lors d'un nouveau projet » ;
  - onglet fichiers : indiquez N:\MAP_C_eleves comme répertoire par défaut.
- Options du compilateur (Menu : Outils – options du compilateur) :
  - onglet options-compilateur C : indiquez « no » pour « support des programmes C norme ANSI standard ».

3 Compilation et exécution d'un programme

Ouvrez le projet **td1_a** :
Menu : Fichiers – "ouvrir projet ou fichier", puis choisir le projet td1_a.dev (sous le sous-répertoire TD1).

Le projet **td1_a** contient le programme suivant :

```
#include <stdio.h>                    //bibliothèque d'entrées/sorties
#include <stdlib.h>
int main()
{
      int i=0, total=0, n=5;
      for(i=1;i<=n;i++)
            total=total+i;
      printf("La somme des entiers de 1 a %d est %d\n",n,total);
      system("pause");          //pour conserver la fenêtre d'exécution ouverte
      return 0;
}
```

Avant d'exécuter ce programme, il faut d'abord le compiler :
Menu : Exécuter – Compiler (raccourci Ctrl-F9).

Corrigez l'erreur de compilation constatée (conseil : regardez les options du projet, onglet compilation : vérifiez le réglage « norme ANSI »).

Exécutez ensuite le programme.

Sans fermer l'environnement de développement "Dev-C++", ouvrez un explorateur Windows et allez sous le répertoire N:\ MAP_C_eleves\TD1\exe qui contient le fichier exécutable td1_a.exe, produit par le projet. Exécutez directement ce programme en double-cliquant sur le fichier exécutable. Revenez à l'environnement de développement et comprenez pourquoi l'exécutable a été placé à cet endroit, en examinant les options du projet.

Etudiez enfin l'utilité de l'avant dernière ligne du programme :

```
system("pause");                //pour conserver ouverte la fenêtre d'exécution
```

Pour cela, procédez comme suit :

- mettez cette ligne en commentaire (en insérant // en première position),
- recompilez,
- exécutez.

4 Créer un projet

Vous allez maintenant construire un projet qui exécute un programme qui affiche le texte "mon premier programme C" dans une fenêtre-console à l'aide de l'instruction suivante :

```
printf("mon premier programme C\n");
```

Pour cela, créez le projet td1_b, toujours sous le sous-répertoire TD1, en respectant les consignes suivantes :

- choisir une application Console,
- choisir un projet C et cocher aussi "langage par défaut".

5 Utiliser le débogueur

Vous allez maintenant apprendre à utiliser le débogueur, à travers la mise au point du programme essai_debug.c dont le code est le suivant :

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
      int somme = 0;
      int deux = 2;
      int sept = '7';
      somme = deux + sept;
      printf("la somme est %d\n", somme);
      system("PAUSE"); return 0;
}
```

Ce programme devrait afficher à l'écran la somme 2+7, soit 9.

Créez, toujours sous le répertoire TD1, le projet td1_c, avec le programme essai_debug.c.

Quand on compile avec succès un programme XXX.c, on obtient du code-objet, dans le fichier XXX.o. Le code objet n'est pas encore exécutable : il faut lui adjoindre le code des fonctions externes appelées (ici printf et system) et établir les liens entre tous ces morceaux de code. Cette opération s'appelle l'**édition de liens**. Elle produit un programme exécutable dans un fichier de suffixe exe (XXX.exe ou PPP.exe, si PPP est le nom du projet).

Quand on veut déboguer un programme, il faut que l'édition de liens ajoute au code des informations supplémentaires pour gérer les opérations d'exécution pas à pas. Faites-le pour le projet td1_c, en modifiant les options du projet :

- onglet "compilateur", choix "édition des liens",
- générer des informations de débogage : OUI

Déclenchez ensuite la compilation puis posez un point d'arrêt sur l'instruction : int deux = 2;

Passez ensuite en exécution avec le débogueur : Menu Debug – Debugger (F8).

L'exécution est déclenchée jusqu'au point d'arrêt. Obtenez l'affichage des variables que vous souhaitez examiner (il suffit de laisser le curseur quelques secondes sur l'identificateur) et exécutez pas à pas pour voir ce qu'il faut corriger. Corrigez l'erreur et re-exécutez le programme.

The GNU C library reference manual :
http://www.gnu.org/software/libc/manual/html_mono/libc.html

Chapitre 2

Programmation avec des fonctions

Sommaire

1 Les fonctions en C

En C, le programme principal et les sous-programmes sont définis comme fonctions. Il n'existe pas de structures spéciales pour le programme principal ni les procédures (comme en Pascal).

Comme dans la plupart des langages, on peut, en C, découper un programme en plusieurs fonctions. Une seule de ces fonctions existe obligatoirement ; c'est la fonction principale appelée `main`. Cette fonction principale peut, éventuellement, appeler une ou plusieurs fonctions secondaires. De même, chaque fonction secondaire peut appeler d'autres fonctions secondaires ou s'appeler elle-même (dans ce dernier cas, on dit que la fonction est *récursive*).

1.1 Qu'est-ce qu'une fonction ?

[Poly], p.58

Une fonction est une sorte de « boîte noire » qui permet d'effectuer une ou plusieurs opérations et que l'on peut utiliser sans se préoccuper de la façon dont sont effectuées ces opérations.

[HK], ch.3, p.105-107

1.2 Définition d'une fonction en C

1.2.1 Définition d'une fonction en langage algorithmique

```
 fonction <NomFonct> (<NomPar1>, <NomPar2>, ...) :<TypeRes>
|    <déclarations des paramètres>
|    <déclarations locales>
|    <instructions>
ffonction
```

1.2.2 Définition d'une fonction en C

La forme générale de la définition de fonction est la suivante :

```
<TypeRés> <NomFonct> (<TypePar1> <NomPar1>,
                      <TypePar2> <NomPar2>, ... )
    {
    <déclarations locales>
    <instructions>
    }
```

—- [HK], ch.3, p.107-108 : Le programme "racine carrée"

Une définition de fonction comprend deux parties :

- un **en-tête** qui précise le type renvoyé par la fonction, le nom (identificateur) de la fonction et ses arguments (paramètres) formels. L'en-tête d'une fonction correspond à son **interface**, c'est-à-dire à ce qui sera visible de l'extérieur ; Cette en-tête a la forme suivante :
  - `<TypeRés>` - le type du résultat de la fonction
  - `<NomFonct>` - le nom de la fonction
  - `<TypePar1> <NomPar1>, <TypePar2> <NomPar2>, ...`, les types et les noms des paramètres de la fonction.
- un **corps** qui contient l'implémentation de la fonction, c'est à dire les instructions à exécuter à chaque appel de la fonction (on parle de *bloc d'instructions* délimité par des accolades { }, contenant :
  - `<déclarations locales>` - les déclarations des données locales (c.-à-d. : des données qui sont uniquement connues à l'intérieur de la fonction)
  - `<instructions>` - la liste des instructions qui définit l'action qui doit être exécutée

```
 TypeRenvoyé NomFonction (Type_1 arg_1, ..., Type_n arg_n)
    {
    instructions
    }
```

Une fonction peut renvoyer ou ne pas renvoyer de résultat, dans ce cas le type renvoyé par la fonction est `void`.

L'instruction `return` permet de renvoyer le résultat de la fonction au programme appelant : `return (expression)`.

Si aucune expression ne suit l'instruction `return`, alors aucune valeur n'est transmise au programme appelant, mais ce dernier reprend la main.

Il faut qu'il y ait correspondance (selon les règles de conversions implicites de types) entre le type du

résultat renvoyé par l'instruction `return` et le type renvoyé par la fonction, qui est déclaré dans son en-tête.

Une fonction peut avoir de 0 à n arguments formels. Les arguments de la fonction sont appelés *paramètres formels*, par opposition aux *paramètres effectifs* qui sont les paramètres avec lesquels la fonction est effectivement appelée. Les paramètres formels peuvent être de n'importe quel type. Leurs identificateurs n'ont d'importance qu'à l'intérieur de la fonction. Enfin, si la fonction ne possède pas de paramètres, on remplace la liste de paramètres formels par le mot-clef `void`.

Une fonction se définit à l'extérieur de toute autre fonction : on ne peut pas emboîter des définitions de fonctions.

Voici quelques exemples de définitions de fonctions :

— Exemple —

```
 int produit (int a, int b)
{
    return(a*b) ;
}

 int puissance (int a, int n)
{
    if (n == 0)
       return(1) ;
    return(a * puissance(a, n-1)) ;        /* on remarquera un appel récursif */
}

 void imprime_tab (int *tab, int nb_elements)
{
    int i ;
    for (i = 0 ; i < nb_elements ; i++)
       printf("%d \t",tab[i]) ;
    printf("\n") ;
    return ;
}
```

1.3 Appel d'une fonction

L'appel d'une fonction se fait par l'expression :

`nom-fonction(para_1, para_2,..., para_n)`

La définition des paramètres (arguments) d'une fonction est placée entre parenthèses ( ) derrière le nom de la fonction. Si une fonction n'a pas besoin de paramètres, les parenthèses restent vides ou contiennent le mot void. La fonction minimale qui ne fait rien et qui ne fournit aucun résultat est alors :

`void dummy() {}`

L'ordre et le type des paramètres effectifs de la fonction doivent concorder avec ceux donnés dans l'en-tête de la fonction. Les paramètres effectifs peuvent être des expressions. La virgule qui sépare deux paramètres effectifs est un simple signe de ponctuation ; il ne s'agit pas de l'opérateur *virgule*. Cela implique en particulier que l'ordre d'évaluation des paramètres effectifs n'est pas assuré et dépend du compilateur. Il est donc déconseillé, pour une fonction à plusieurs paramètres, de faire figurer des opérateurs d'incrémentation ou de décrémentation (++ ou −) dans les expressions définissant les paramètres effectifs.

1.4 Résultat d'une fonction

Par définition, **toute fonction en C fournit un résultat dont le type doit être défini**. Si aucun type n'est défini explicitement, C suppose par défaut que le type du résultat est int (integer).

Le retour du résultat se fait en général à la fin de la fonction par l'instruction `return`.

Le type d'une fonction qui ne fournit pas de résultat (comme les procédures en langage algorithmique ou en Pascal), est déclaré comme `void` (vide).

1.5 Déclaration d'une fonction

Le C n'autorise pas les fonctions imbriquées. La définition d'une fonction secondaire doit donc être placée soit avant, soit après la fonction principale `main`. Toutefois, il est indispensable que le compilateur "connaisse" la fonction au moment où celle-ci est appelée. Si une fonction est définie après son premier appel (en particulier si sa définition est placée après la fonction `main`), elle doit impérativement être déclarée au préalable. Une fonction secondaire est déclarée par son prototype, qui donne le type de la fonction et celui de ses paramètres, sous la forme :

```
type nom_fonction (type_1, ..., type_n) ;
```

Les fonctions secondaires peuvent être déclarées indifféremment avant ou au début de la fonction `main`. Par exemple, on écrira

— Exemple —

```
 int puissance (int, int ) ;              /* Déclaration de la fonction */

int puissance (int a, int n)             /* Définition de la fonction */
{
    if (n == 0)
       return(1) ;
    return(a * puissance(a, n-1)) ;
}

main()
{     int a = 2, b = 5 ;
    printf("%d\n", puissance(a,b)) ;        /* Appel de la fonction */
}
```

Même si la déclaration est parfois facultative (par exemple quand les fonctions sont définies avant la fonction `main` et dans le bon ordre), elle seule permet au compilateur de vérifier que le nombre et le type des paramètres utilisés dans la définition concordent bien avec le protype. De plus, la présence d'une déclaration permet au compilateur de mettre en place d'éventuelles conversions des paramètres effectifs, lorsque la fonction est appelée avec des paramètres dont les types ne correspondent pas aux types indiqués dans le prototype.

Ainsi les fichiers d'extension .h de la librairie standard (fichiers headers) contiennent notamment les prototypes des fonctions de la librairie standard. Par exemple, on trouve dans le fichier `math.h` le prototype de la fonction `pow` (élévation à la puissance) :

```
extern double pow(double , double) ;
```

La directive au préprocesseur `#include <math.h>`

permet au préprocesseur d'inclure la déclaration de la fonction `pow` dans le fichier source. Ainsi, si cette fonction est appelée avec des paramètres de type `int`, ces paramètres seront convertis en `double` lors de la compilation.

Par contre, en l'absence de directive au préprocesseur, le compilateur ne peut effectuer la conversion de type. Dans ce cas, l'appel à la fonction pow avec des paramètres de type `int` peut produire un résultat faux !

1.6 Instructions

En C, toute instruction simple est terminée par un point-virgule ; (même si elle se trouve en dernière position dans un bloc d'instructions). Par exemple :

```
printf("hello, world\n") ;
```

1.7 La portée d'une fonction et sa déclaration

[Poly], p.61

Une fonction est nécessairement définie à l'extérieur de tout bloc d'instructions. Elle est un **objet global**.

La portée d'une fonction définie dans un fichier s'étend de l'endroit où elle est définie à la fin du fichier.

La portée d'une fonction peut être étendue à tout fichier dans lequel elle est déclarée. De ce fait une fonction en C est un **objet global externe**.

EXEMPLE

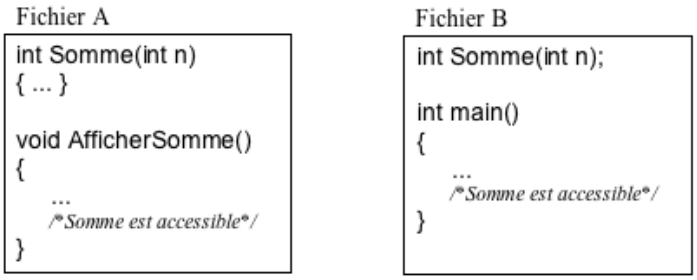


FIG. 2.1: Exemple.

La fonction `Somme` définie dans le fichier A peut être utilisée (appelée) dans le fichier B. Le fichier B n'a pas besoin de connaître l'implémentation de la fonction `Somme`, mais il doit contenir une déclaration de cette fonction.

2 La durée de vie des variables

Les variables manipulées dans un programme C ne sont pas toutes traitées de la même manière. En particulier, elles n'ont pas toutes la même durée de vie. On distingue deux catégories de variables.

Les variables permanentes (ou statiques)

Une variable permanente occupe un emplacement en mémoire qui reste le même durant toute l'exécution du programme. Cet emplacement est alloué une fois pour toutes lors de la compilation. La partie de la mémoire contenant les variables permanentes est appelée *segment de données*. Par défaut, les variables permanentes sont initialisées à zéro par le compilateur. Elles sont caractérisées par le mot-clef `static`.

Les variables temporaires

Les variables temporaires se voient allouer un emplacement en mémoire de façon dynamique lors de l'exécution du programme. Elles ne sont pas initialisées par défaut. Leur emplacement en mémoire est libéré par exemple à la fin de l'exécution d'une fonction secondaire.

Par défaut, les variables temporaires sont situées dans la partie de la mémoire appelée *segment de pile*. Dans ce cas, la variable est dite *automatique*. Le spécificateur de type correspondant, auto, est rarement utilisé puisqu'il ne s'applique qu'aux variables temporaires qui sont automatiques par défaut.

La durée de vie des variables est liée à leur *portée*, c'est-à-dire à la portion du programme dans laquelle elles sont définies.

2.1 Variables globales

On appelle *variable globale* une variable déclarée en dehors de toute fonction. Une variable globale est connue du compilateur dans toute la portion de code qui suit sa déclaration. Les variables globales sont systématiquement permanentes. Dans le programme suivant, `n` est une variable globale :

EXEMPLE

```
int n ;
void fonction() ;

void fonction()
{
    n++ ;
    printf("appel numero %d\n", n) ;
    return ;
}

main()
{
    int i ;
    for (i = 0 ; i < 5 ; i++)
       fonction() ;
}
```

La variable `n` est initialisée à zéro par le compilateur et il s'agit d'une variable permanente. En effet, le programme affiche :

```
appel numero 1
appel numero 2
appel numero 3
appel numero 4
appel numero 5
```

2.2 Variables locales

On appelle variable locale une variable déclarée à l'intérieur d'une fonction (ou d'un bloc d'instructions) du programme. Par défaut, les variables locales sont temporaires. Quand une fonction est appelée, elle place ses variables locales dans la pile. A la sortie de la fonction, les variables locales sont dépilées et donc perdues.

Les variables locales n'ont en particulier aucun lien avec des variables globales de même nom. Par exemple, le programme suivant :

— Exemple —

```
int n = 10 ;              /* Variable globale */
void fonction() ;

void fonction()
{
    int n = 0 ;              /* Variable locale */
    n++ ;
    printf("appel numero %d\n",n) ;
    return ;
}

main()
{
    int i ;
    for (i = 0 ; i < 5 ; i++)
       fonction() ;
}
```

affiche :

```
appel numero 1
appel numero 1
appel numero 1
appel numero 1
appel numero 1
```

Les variables locales à une fonction ont une durée de vie limitée à une seule exécution de cette fonction. Leurs valeurs ne sont pas conservées d'un appel au suivant.

Il est toutefois possible de créer une variable locale de classe statique en faisant précéder sa déclaration du mot-clef `static` :

```
static type nom_de_variable ;
```

Une telle variable reste locale à la fonction dans laquelle elle est déclarée, mais sa valeur est conservée d'un appel au suivant. Elle est également initialisée à zéro à la compilation. Par exemple, dans le programme suivant, `n` est une variable locale à la fonction secondaire `fonction`, mais de classe statique.

EXEMPLE

```
int n = 10 ;
void fonction() ;

void fonction()
{
    static int n ;
    n++ ;
    printf("appel numero %d\n",n) ;
    return ;
}

main()
{
    int i ;
    for (i = 0 ; i < 5 ; i++)
       fonction() ;
}
```

Ce programme affiche :

```
appel numero 1
appel numero 2
appel numero 3
appel numero 4
appel numero 5
```

On voit que la variable locale `n` est de classe statique (elle est initialisée à zéro, et sa valeur est conservée d'un appel à l'autre de la fonction). Par contre, il s'agit bien d'une variable locale, qui n'a aucun lien avec la variable globale du même nom.

2.3 La transmission des arguments d'une fonction "par valeur"

[Poly], p.62

Les paramètres d'une fonction sont traités de la même manière que les variables locales de classe automatique : lors de l'appel de la fonction, les paramètres effectifs sont copiés dans le segment de pile. La fonction travaille alors uniquement sur cette copie. Cette copie disparaît lors du retour au programme appelant. Cela implique en particulier que, si la fonction modifie la valeur d'un de ses paramètres, seule la copie sera modifiée ; la variable du programme appelant, elle, ne sera pas modifiée. On dit que les paramètres d'une fonction sont *transmis par valeurs*.

On appelle **arguments formels** les arguments figurant dans l'en-tête de la définition d'une fonction, et **arguments effectifs** les arguments fournis lors de l'appel de la fonction.

Une fonction ne peut pas modifier la valeur d'un argument effectif. En effet, la fonction reçoit une copie de la valeur de l'argument effectif et toute modification effectuées sur cette copie n'a aucune incidence sur la valeur de l'argument effectif.

Exemple

```
#include<stdio.h>

void Echange(int a, int b)
{
    int temp=0 ;
    temp = a ; a = b ; b = temp ;
}

int main()
{
   int n=2, p=5 ;
   Echange(n, p) ;
   printf("n=%d \t p=%d \n",n,p) ;
   system("pause") ; return 0 ;
}
```

L'appel de la fonction `Echange` dans la fonction `main` laisse les arguments effectifs `n` et`p` inchangés.

2.4 La transmission des arguments d'une fonction "par adresse"

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*.

Pour qu'une fonction modifie la valeur d'un de ses arguments, il faut qu'elle ait pour paramètre l'adresse de cet objet et non sa valeur. Par exemple, pour échanger les valeurs de deux variables, il faut écrire :

Exemple

```
void echange (int *, int *) ;

void echange (int *adr_a, int *adr_b)
{
int t ;
t = *adr_a ;
*adr_a = *adr_b ;
*adr_b = t ;
return ;
}

main()
{
    int a = 2, b = 5 ;
    printf("debut programme principal : \n a = %d \t b = %d\n",a,b) ;
    echange(&a,&b) ;
    printf("fin programme principal : \n a = %d \t b = %d\n",a,b) ;
}
```

Autre 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 :

Exemple

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

```
{
*c = a + b ;
}

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

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

Nous verrons au chapitre 3 qu'un tableau est un pointeur (sur le premier élément du tableau). Lorsqu'un tableau est transmis comme paramètre à une fonction secondaire, ses éléments sont donc modifiés par la fonction. Par exemple, le programme :

EXEMPLE

```
#include <stdlib.h>

void init (int *, int ) ;

void init (int *tab, int n)
{
    int i ;
    for (i = 0 ; i < n ; i++)
       tab[i] = i ;
    return ;
}

main()
{
    int i, n = 5 ;
    int *tab ;
    tab = (int*)malloc(n * sizeof(int)) ;
    init(tab,n) ;
}
```

---

initialise les éléments du tableau `tab`.

2.5 Les qualificateurs de type `const` et `volatile`

Les qualificateurs de type `const` et `volatile` permettent de réduire les possibilités de modifier une variable.

`const`

Une variable dont le type est qualifié par `const` ne peut pas être modifiée. Ce qualificateur est utilisé pour se protéger d'une erreur de programmation. On l'emploie principalement pour qualifier le type des paramètres d'une fonction afin d'éviter de les modifier involontairement.

`volatile`

Une variable dont le type est qualifié par `volatile` ne peut pas être impliquée dans les optimisations effectuées par le compilateur. On utilise ce qualificateur pour les variables susceptibles d'être modifiées par une action extérieure au programme.

Les qualificateurs de type se placent juste avant le type de la variable, par exemple

```
const char c ;
```

désigne un caractère non modifiable.

2.6 La fonction main

La fonction principale `main` est une fonction comme les autres. Nous avons jusqu'à présent considéré qu'elle était de type `void`, ce qui est toléré par le compilateur. Toutefois l'écriture

```
main()
```

provoque un message d'avertissement lorsqu'on utilise l'option `-Wall` de `gcc` :

```
% gcc -Wall prog.c
prog.c :5 : warning : return-type defaults to ‘int’
prog.c : In function ‘main’ :
prog.c :11 : warning : control reaches end of non-void function
```

En fait, la fonction `main` est de type `int`. Elle doit retourner un entier dont la valeur est transmise à l'environnement d'exécution. Cet entier indique si le programme s'est ou non déroulé sans erreur. La valeur de retour 0 correspond à une terminaison correcte, toute valeur de retour non nulle correspond à une terminaison sur une erreur. On peut utiliser comme valeur de retour les deux constantes symboliques `EXIT_SUCCESS` (égale à 0) et `EXIT_FAILURE` (égale à 1) définies dans `stdlib.h`.

L'instruction `return(statut) ;` dans la fonction `main`, où `statut` est un entier spécifiant le type de terminaison du programme, peut être remplacée par un appel à la fonction `exit` de la librairie standard (`stdlib.h`). La fonction `exit`, de prototype

```
void exit(int statut) ;
```

provoque une terminaison normale du programme en notifiant un succès ou un échec selon la valeur de l'entier `statut`.

Lorsqu'elle est utilisée sans arguments, la fonction `main` a donc pour prototype

```
int main(void) ;
```

La fonction `main` peut également posséder des paramètres formels. En effet, un programme C peut recevoir une liste d'arguments au lancement de son exécution. La ligne de commande qui sert à lancer le programme est, dans ce cas, composée du nom du fichier exécutable suivi par des paramètres. La fonction `main` reçoit tous ces éléments de la part de l'interpréteur de commandes.

En fait, la fonction `main` possède deux paramètres formels, appelés par convention `argc` (argument count) et `argv` (argument vector).

`argc` est une variable de type `int` dont la valeur est égale au nombre de mots composant la ligne de commande (y compris le nom de l'exécutable). Elle est donc égale au nombre de paramètres effectifs de la fonction + 1.

`argv` est un tableau de chaînes de caractères correspondant chacune à un mot de la ligne de commande. Le premier élément `argv[0]` contient donc le nom de la commande (du fichier exécutable), le second `argv[1]` contient le premier paramètre....

Le second prototype valide de la fonction `main` est donc

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

— Exemple —

Ainsi, le programme suivant calcule le produit de deux entiers, entrés en arguments de l'exécutable :

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

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

```
int a, b ;

if (argc != 3)
{
       printf("\nErreur : nombre invalide d'arguments") ;
       printf("\nUsage : %s int int\n",argv[0]) ;
       return(EXIT_FAILURE) ;
}
    a = atoi(argv[1]) ;
    b = atoi(argv[2]) ;
    printf("\n Le produit de %d par %d vaut :     return(EXIT_SUCCESS) ;
}
```

On lance donc l'exécutable avec deux paramètres entiers, par exemple,
`a.out 12 8`
Ici, `argv` sera un tableau de 3 chaînes de caractères `argv[0]`, `argv[1]` et `argv[2]` qui, dans notre exemple, valent respectivement `"a.out"`, `"12"` et `"8"`.

Enfin, la fonction de la librairie standard `atoi()`, déclarée dans `stdlib.h`, prend en argument une chaîne de caractères et retourne l'entier dont elle est l'écriture décimale.

2.7 Fonctions avec un nombre variable de paramètres

Il est possible en C de définir des fonctions qui ont un nombre variable de paramètres. En pratique, il existe souvent des méthodes plus simples pour gérer ce type de problème : toutefois, cette fonctionnalité est indispensable dans certains cas, notamment pour les fonctions `printf` et `scanf`.

Une fonction possédant un nombre variable de paramètre doit posséder au moins un paramètre formel fixe. La notation ...(obligatoirement à la fin de la liste des paramètres d'une fonction) spécifie que la fonction possède un nombre quelconque de paramètres (éventuellement de types différents) en plus des paramètres formels fixes. Ainsi, une fonction ayant pour prototype

`int f(int a, char c, ...) ;`

prend comme paramètre un entier, un caractère et un nombre quelconque d'autres paramètres.

De même le prototype de la fonction `printf` est :

`int printf(char *format, ...) ;`

puisque `printf` a pour argument une chaîne de caractères spécifiant le format des données à imprimer, et un nombre quelconque d'autres arguments qui peuvent être de types différents.

Un appel à une fonction ayant un nombre variable de paramètres s'effectue comme un appel à n'importe quelle autre fonction.

Pour accéder à la liste des paramètres de l'appel, on utilise les macros définies dans le fichier en-tête `stdarg.h` de la librairie standard. Il faut tout d'abord déclarer dans le corps de la fonction une variable pointant sur la liste des paramètres de l'appel ; cette variable a pour type `va_list`. Par exemple,

`va_list liste_parametres ;`

Cette variable est tout d'abord initialisée à l'aide de la macro `va_start`, dont la syntaxe est

`va_start(liste_parametres, dernier_parametre) ;`

où `dernier_parametre` désigne l'identificateur du dernier paramètre formel fixe de la fonction. Après traitement des paramètres, on libère la liste à l'aide de la `va_end` :

`va_end(liste_parametres) ;`

On accède aux différents paramètres de liste par la macro `va_arg` qui retourne le paramètre suivant de la liste :

`va_arg(liste_parametres, type)`

où `type` est le type supposé du paramètre auquel on accède.

Notons que l'utilisateur doit lui-même gérer le nombre de paramètres de la liste. Pour cela, on utilise généralement un paramètre formel qui correspond au nombre de paramètres de la liste, ou une valeur particulière qui indique la fin de la liste.

Cette méthode est utilisée dans le programme suivant, où la fonction `add` effectue la somme de ses paramètres en nombre quelconque.

Exemple

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

int add(int,...) ;

int add(int nb,...)
{
    int res = 0 ;
    int i ;
    va_list liste_parametres ;

    va_start(liste_parametres, nb) ;
    for (i = 0 ; i < nb ; i++)
       res += va_arg(liste_parametres, int) ;
    va_end(liste_parametres) ;
    return(res) ;
}

int main(void)
{
    printf("\n %d", add(4,10,2,8,5)) ;
    printf("\n %d \n", add(6,10,15,5,2,8,10)) ;
    return(EXIT_SUCCESS) ;
}
```

2.8 Espace mémoire associé aux fonctions

[HK] : ch.3 ; p.134-135

2.9 Avantages d'utiliser des fonctions

[HK] : ch.3 ; p.122

- Abstraction procédurale
- Réutilisation de code

2.10 Règles de bonne conduite : pré-conditions et post-conditions

[HK] : ch.3 ; p.130-131

2.11 Éviter les erreurs de programmation

[HK] : ch.3 ; p.136

3 Les fonctions prédéfinies : Les librairies C de fonctions

[HK] : ch.3 ; p.105-110

4 Travaux dirigés

[Poly], p.18 : exemple de programme de traitement de chaîne de caractères

[Poly], p.20 : programme à étendre.

Exercices

Programmer la fonction de Fibonacci de manière récursive et de manière itérative.
[C. Delannoy, "exos"], ex-30, p.32-33 (**10mn**)
[C. Delannoy, "exos"], ex-31, p.33-34 (**10mn**) (sur la portée des variables)
[C. Delannoy, "exos"], ex-35, p.37-38 (**20mn**) (lit un entier et précise si multiple de 2 ou de 3)
[C. Delannoy, "exos"], ex-34, p.36-37 (**10mn**) (Compte du nb de fois où la fonction est appelée)
[C. Delannoy, "exos"], ex-32, p.34-35 (**20mn**) (opération appelée suivant l'opérateur)

Chapitre 3

Les types de données composés

Sommaire

À partir des types prédéfinis du C (caractères, entiers, flottants), on peut créer de nouveaux types, appelés types composés, qui permettent de représenter des ensembles de données organisées.

1 Les tableaux

Un tableau est un ensemble fini d'éléments de même type, stockés en mémoire à des adresses contiguës.

La déclaration d'un tableau à une dimension se fait de la façon suivante :

```
type nom-du-tableau[nombre-éléments] ;
```

où `nombre-éléments` est une expression constante entière positive.

Par exemple, la déclaration `int tab[10] ;` indique que `tab` est un tableau de 10 éléments de type `int`. Cette déclaration alloue donc en mémoire pour l'objet `tab` un espace de 10×4 octets consécutifs.

Pour plus de clarté, il est recommandé de donner un nom à la constante nombre-éléments par une directive au préprocesseur, par exemple

```
#define nombre-éléments 10
```

On accède à un élément du tableau en lui appliquant l'opérateur `[]`. Les éléments d'un tableau sont toujours numérotés de 0 à `nombre-éléments -1`. Le programme suivant imprime les éléments du tableau `tab` :

Exemple

```
#define N 10
main()
{
int tab[N] ;
int i ;
...
for (i = 0 ; i < N ; i++)
printf("tab[%d] = %d\n",i,tab[i]) ;
}
```

Un tableau correspond en fait à un pointeur vers le premier élément du tableau. Ce pointeur est constant. Cela implique en particulier qu'aucune opération globale n'est autorisée sur un tableau. Notamment, un tableau ne peut pas figurer à gauche d'un opérateur d'affectation. Par exemple, on ne peut pas écrire "`tab1 = tab2 ;`". Il faut effectuer l'affectation pour chacun des éléments du tableau :

— Exemple —

```
#define N 10
main()  {
    int tab1[N], tab2[N] ;
    int i ;
    ...
    for (i = 0 ; i < N ; i++)
       tab1[i] = tab2[i] ;
}
```

On peut initialiser un tableau lors de sa déclaration par une liste de constantes de la façon suivante :

`type nom-du-tableau[N] = {constante-1,constante-2,...,constante-N} ;`

Par exemple, on peut écrire

— Exemple —

```
#define N 4
int tab[N] = {1, 2, 3, 4} ;
main()
{
    int i ;
    for (i = 0 ; i < N ; i++)
       printf("tab[%d] = %d\n",i,tab[i]) ;
}
```

Si le nombre de données dans la liste d'initialisation est inférieur à la dimension du tableau, seuls les premiers éléments seront initialisés. Les autres éléments seront mis à zéro si le tableau est une variable globale (extérieure à toute fonction) ou une variable locale de classe de mémorisation `static` (cf. chapitre 4).

De la même manière un tableau de caractères peut être initialisé par une liste de caractères, mais aussi par une chaîne de caractères littérale. Notons que le compilateur complète toute chaîne de caractères avec un caractère nul `'\0'`. Il faut donc que le tableau ait au moins un élément de plus que le nombre de caractères de la chaîne littérale.

— Exemple —

```
#define N 8
```

```
char tab[N] = "exemple" ;
main()
{
    int i ;
    for (i = 0 ; i < N ; i++)
       printf("tab[%d] = %c\n",i,tab[i]) ;
}
```

Lors d'une initialisation, il est également possible de ne pas spécifier le nombre d'éléments du tableau. Par défaut, il correspondra au nombre de constantes de la liste d'initialisation. Ainsi le programme suivant imprime le nombre de caractères du tableau `tab`, ici 8.

— Exemple —

```
char tab[] = "exemple" ;
main()
{
    int i ;
    printf("Nombre de caracteres du tableau = %d\n",sizeof(tab)/sizeof(char)) ;
}
```

Remarque : La fonction prédéfinie C `sizeof` est applicable à tout type de données, prédéfinis ou définis par l'utilisateur, et renvoie le nombre d'octets utilisés à l'instant courant pour son stockage.

De manière similaire, on peut déclarer un **tableau à plusieurs dimensions**. Par exemple, pour un tableau à deux dimensions :

`type nom-du-tableau[nombre-lignes][nombre-colonnes]`

En fait, un tableau à deux dimensions est un tableau unidimensionnel dont chaque élément est lui-même un tableau. On accède à un élément du tableau par l'expression "`tableau[i][j]`". Pour initialiser un tableau à plusieurs dimensions à la compilation, on utilise une liste dont chaque élément est une liste de constantes :

— Exemple —

```
#define M 2
#define N 3
int tab[M][N] = {{1, 2, 3}, {4, 5, 6}} ;

main()
{
    int i, j ;
    for (i = 0 ; i < M ; i++)
       {
       for (j = 0 ; j < N ; j++)
          printf("tab[%d][%d]=%d\n",i,j,tab[i][j]) ;
}
}
```

2 Les structures

Nous avons vu comment le tableau permet de désigner sous un seul nom un ensemble de valeurs de même type, chacune d'entre elles étant repérée par un indice.

La structure, quant à elle, permet de désigner sous un seul nom un ensemble de valeurs pouvant être de types différents. L'accès à chaque élément de la structure (nomm *champ*) se fait, cette fois, non plus par une indication de position, mais par son nom au sein de la structure.

Une structure est une suite finie d'objets de types différents. Contrairement aux tableaux, les différents éléments d'une structure n'occupent pas nécessairement des zones contiguës en mémoire. Chaque élément de la structure, appelé *membre* ou *champ*, est désigné par un identificateur.

On distingue la déclaration d'un *modèle de structure* de celle d'un objet de type structure correspondant à un modèle donné. La déclaration d'un modèle de structure dont l'identificateur est modele suit la syntaxe suivante :

```
struct modele
{type-1 membre-1 ;
type-2 membre-2 ;
...
type-n membre-n ;
} ;
```

Pour déclarer un objet de type structure correspondant au modèle précédent, on utilise la syntaxe :

```
struct modele objet ;
```

ou bien, si le modèle n'a pas été déclaré au préalable :

```
struct modele
{type-1 membre-1 ;
type-2 membre-2 ;
...
type-n membre-n ;
}objet ;
```

On accède aux différents membres d'une structure grâce à l'opérateur membre de structure, noté ".". Le i-ème membre de objet est désigné par l'expression

```
objet.membre-i
```

On peut effectuer sur le i-ème membre de la structure toutes les opérations valides sur des données de type {type-i}.

Par exemple, le programme suivant définit la structure complexe, composée de deux champs de type `double` ; il calcule la norme d'un nombre complexe.

— Exemple —

```
#include <math.h>
struct complexe
{
    double reelle ;
    double imaginaire ;
} ;

main()
{
    struct complexe z ;
    double norme ;
    ...
    norme = sqrt(z.reelle * z.reelle + z.imaginaire * z.imaginaire) ;
    printf("norme de (%f + i %f) = %f \n",z.reelle,z.imaginaire,norme) ;
}
```

Les règles d'initialisation d'une structure lors de sa déclaration sont les mêmes que pour les tableaux. On écrit par exemple :

```
struct complexe z = 2. , 2. ;
```

En ANSI C, on peut appliquer l'opérateur d'affectation aux structures (à la différence des tableaux). Dans le contexte précédent, on peut écrire :

EXEMPLE

```
...
main()
{
    struct complexe z1, z2 ;
    ...
    z2 = z1 ;
}
```

2.1 Les énumérations

Le type énumération est un cas particulier de type entier. Sa déclaration et son utilisation sont très proches du type structure.

Les énumérations permettent de définir un type par la liste des valeurs qu'il peut prendre. Un objet de type énumération est défini par le mot-clef `enum` et un identificateur de modèle, suivis de la liste des valeurs que peut prendre cet objet :

```
enum modele constante-1, constante-2,...,constante-n ;
```

En réalité, les objets de type enum sont représentés comme des `int`. Les valeurs possibles `constante-1, constante-2,...,constante-n` sont codées par des entiers de `0` à `n-1`.

Par exemple, le type `enum booleen` défini dans le programme suivant associe l'entier `0` à la valeur `faux` et l'entier `1` à la valeur `vrai`.

EXEMPLE

```
main()
{
    enum booleen {faux, vrai} ;
    enum booleen b ;
    b = vrai ;
    printf("b = %d\n",b) ;
}
```

On peut modifier le codage par défaut des valeurs de la liste lors de la déclaration du type énuméré, par exemple :

```
enum booleen faux = 12, vrai = 23 ;
```

2.2 Définition de types composés avec `typedef`

Pour alléger l'écriture des programmes, on peut affecter un nouvel identificateur à un type composé à l'aide de `typedef` :

```
typedef type synonyme ;
```

Par exemple,

EXEMPLE

```
struct complexe
{       double reelle ;
```

```
    double imaginaire ;
} ;
typedef struct complexe complexe ;

main()
{
    complexe z ;
    ...
}
```

2.3 Les champs de bits

2.4 Les unions

2.5 Un peu d'histoire

Exemple 1

- insertion ou suppression de Paul Brasse, mauvais nageur, qui a pris un bain de 2 minutes le 14/07/1989 sur la plage de sable très polluée de Binic, en Bretagne.
- recherche des noms des nageurs ayant pris des bains de plus d'une minute en Bretagne.
- suppression de tous les nageurs ayant pris, en février, des bains de plus de 2 minutes en Bretagne (hydrocution ?).

Pour manipuler ces relations, nous avons besoin d'un langage adapté dont la particularité est de savoir manipuler aisément ces tableaux de données. Ce langage constitue l'algèbre relationnelle.

Exemple 2

Toutes les dépendances fonctionnelles citées précédemment peuvent être représentées comme sur la figure 3.1.

FIG. 3.1: Graphe de dépendances fonctionnelles.

Chapitre 4

Les pointeurs

Sommaire

Toute variable manipulée dans un programme est stockée quelque part en mémoire centrale. Cette mémoire est constituée d'octets qui sont identifiés de manière univoque par un numéro qu'on appelle *adresse*. Pour retrouver une variable, il suffit donc de connaître l'adresse de l'octet où elle est stockée (ou, s'il s'agit d'une variable qui recouvre plusieurs octets contigus, l'adresse du premier de ces octets). Pour des raisons évidentes de lisibilité, on désigne souvent les variables par des identificateurs, et non par leur adresse. C'est le compilateur qui fait alors le lien entre l'identificateur d'une variable et son adresse en mémoire. Toutefois, il est parfois très pratique de manipuler directement une variable par son adresse.

1 Adresse et valeur d'un objet

On appelle *lvalue* (*left value*) tout objet pouvant être placé à gauche d'un opérateur d'affectation. Une *lvalue* est caractérisée par :

- son adresse, c'est-à-dire l'adresse-mémoire à partir de laquelle l'objet est stocké ;
- sa valeur, c'est-à-dire ce qui est stocké à cette adresse.

Dans l'exemple,

```
int i, j ;
i = 3 ;
j = i ;
```

Si le compilateur a placé la variable `i` à l'adresse 4831836000 en mémoire, et la variable `j` à l'adresse 4831836004, on a :

objet	adresse	valeur
`i`	4831836000	3
`j`	4831836004	3

Deux variables différentes ont des adresses différentes. L'affectation `j = i ;` n'opère que sur les valeurs des variables. Les variables `i` et `j` étant de type `int`, elles sont stockées sur 4 octets. Ainsi la valeur de `i` est stockée sur les octets d'adresse 4831836000 à 4831836003.

L'adresse d'un objet étant un numéro d'octet en mémoire, il s'agit d'un entier quelque soit le type de l'objet considéré. Le format interne de cet entier (16 bits, 32 bits ou 64 bits) dépend des architectures. Sur un DEC alpha, par exemple, une adresse a toujours le format d'un entier long (64 bits).

L'opérateur `&` permet d'accéder à l'adresse d'une variable. Toutefois `&i` n'est pas une *lvalue* mais une constante : on ne peut pas faire figurer `&i` à gauche d'un opérateur d'affectation. Pour pouvoir manipuler des adresses, on doit donc recourir un nouveau type d'objets, les pointeurs.

2 Notion de pointeurs

Un pointeur est un objet (*lvalue*) dont la valeur est égale à l'adresse d'un autre objet. On déclare un pointeur par l'instruction :

```
type *nom-du-pointeur ;
```

où `type` est le type de l'objet pointé. Cette déclaration déclare un identificateur, `nom-du-pointeur`, associé à un objet dont la valeur est l'adresse d'un autre objet de type `type`. L'identificateur `nom-du-pointeur` est donc en quelque sorte un identificateur d'adresse. Comme pour n'importe quelle *lvalue*, sa valeur est modifiable.

Même si la valeur d'un pointeur est toujours un entier (éventuellement un entier long), le type d'un pointeur dépend du type de l'objet vers lequel il pointe. Cette distinction est indispensable à l'interprétation de la valeur d'un pointeur. En effet, pour un pointeur sur un objet de type `char`, la valeur donne l'adresse de l'octet où cet objet est stocké. Par contre, pour un pointeur sur un objet de type `int`, la valeur donne l'adresse du premier des 4 octets où l'objet est stocké. Dans l'exemple suivant, on définit un pointeur `p` qui pointe vers un entier `i` :

```
int i = 3 ;
int *p ;
p = &i ;
```

On se trouve dans la configuration :

objet	adresse	valeur
`i`	4831836000	3
`p`	4831836004	4831836000

L'opérateur unaire d'indirection `*` permet d'accéder directement à la valeur de l'objet pointé. Ainsi, si `p` est un pointeur vers un entier `i`, `*p` désigne la valeur de `i`. Par exemple, le programme

— Exemple —

```
main()
```

```
{
    int i = 3 ;
    int *p ;

    p = &i ;                        /* p = adresse de i */
    printf("*p = %d \n",*p) ;       /* ce qui est pointé par p */
}
```

imprime `*p = 3`.

Dans ce programme, les objets `i` et `*p` sont identiques : ils ont mêmes adresse et valeur. Nous sommes dans la configuration :

objet	adresse	valeur
`i`	4831836000	3
`p`	4831836004	4831836000
`*p`	4831836000	3

Cela signifie en particulier que toute modification de `*p` modifie `i`. Ainsi, si l'on ajoute l'instruction `*p = 0 ;` à la fin du programme précédent, la valeur de `i` devient nulle.

On peut donc dans un programme manipuler à la fois les objets `p` et `*p`. Ces deux manipulations sont très différentes. Comparons par exemple les deux programmes suivants :

— Exemple —

```
main()
{
    int i = 3, j = 6 ;
    int *p1, *p2 ;
    p1 = &i ;
    p2 = &j ;
    *p1 = *p2 ;  }
```

et

```
main()
{
    int i = 3, j = 6 ;
    int *p1, *p2 ;
    p1 = &i ;
    p2 = &j ;
    p1 = p2 ;
}
```

Avant la dernière affectation de chacun de ces programmes, on est dans une configuration du type :

objet	adresse	valeur
`i`	4831836000	3
`j`	4831836004	6
`p1`	4831835984	4831836000
`p2`	4831835992	4831836004

Après l'affectation `*p1 = *p2 ;` du premier programme, on a

objet	adresse	valeur
`i`	4831836000	6
`j`	4831836004	6
`p1`	4831835984	4831836000
`p2`	4831835992	4831836004

Par contre, l'affectation `p1 = p2` du second programme, conduit à la situation :

objet	adresse	valeur
`i`	4831836000	3
`j`	4831836004	6
`p1`	4831835984	4831836004
`p2`	4831835992	4831836004

3 Opérations sur les pointeurs

La valeur d'un pointeur étant un entier, on peut lui appliquer un certain nombre d'opérateurs arithmétiques classiques.

Les seules **opérations arithmétiques** valides sur les pointeurs sont :
- l'*addition d'un entier à un pointeur*. Le résultat est un pointeur de même type que le pointeur de départ ;
- la *soustraction d'un entier à un pointeur*. Le résultat est un pointeur de même type que le pointeur de départ ;
- la *différence de deux pointeurs* pointant tous deux vers des objets de même type. Le résultat est un entier.

Notons que la somme de deux pointeurs n'est pas autorisée.

Si `i` est un entier et `p` est un pointeur sur un objet de type `type`, l'expression `p + i` désigne un pointeur sur un objet de type `type` dont la valeur est égale à la valeur de `p` incrémentée de `i * sizeof(type)`. Il en va de même pour la soustraction d'un entier à un pointeur, et pour les opérateurs d'incrémentation et de décrémentation `++` et `--`.

Si `p` et `q` sont deux pointeurs sur des objets de type `type`, l'expression `p - q` désigne un entier dont la valeur est égale à `(p - q)/sizeof(type)` .

Par exemple, le programme :

Exemple

```
main()
{
    int i = 3 ;
    int *p1, *p2 ;
    p1 = &i ;
    p2 = p1 + 1 ;
    printf("p1 = %ld \t p2 = %ld\n",p1,p2) ;
}
```

affiche `p1 = 4831835984 p2 = 4831835988`.

Par contre, le même programme avec des pointeurs sur des objets de type `double` :

Exemple

```
main()
{
    double i = 3 ;
    double *p1, *p2 ;
    p1 = &i ;
    p2 = p1 + 1 ;
    printf("p1 = %ld \t p2 = %ld\n",p1,p2) ;
}
```

affiche `p1 = 4831835984 p2 = 4831835992`.

Les **opérateurs de comparaison** sont également applicables aux pointeurs, à condition de comparer des pointeurs qui pointent vers des objets de même type.

L'utilisation des opérations arithmétiques sur les pointeurs est particulièrement utile pour parcourir des tableaux. Ainsi, le programme suivant imprime les éléments du tableau tab dans l'ordre croissant puis décroissant des indices.

— Exemple —

```
#define N 5
int tab[5] = {1, 2, 6, 0, 7};
main()
{
    int *p;
    printf("\n ordre croissant :\n");
    for (p = &tab[0]; p <= &tab[N-1]; p++)
       printf(" %d \n",*p);
    printf("\n ordre decroissant :\n");
    for (p = &tab[N-1]; p >= &tab[0]; p--)
       printf(" %d \n",*p);
}
```

4 Allocation dynamique

Avant de manipuler un pointeur, et notamment de lui appliquer l'opérateur d'indirection `*`, il faut l'initialiser. Sinon, par défaut, la valeur du pointeur est égale à une constante symbolique notée `NULL` définie dans `stdio.h`. En général, cette constante vaut `0`. Le test `p == NULL` permet de savoir si le pointeur `p` pointe vers un objet.

On peut initialiser un pointeur `p` par une affectation sur `p`. Par exemple, on peut affecter à `p` l'adresse d'une autre variable. Il est également possible d'affecter directement une valeur à `*p`. Mais pour cela, il faut d'abord réserver à `*p` un espace-mémoire de taille adéquate. L'adresse de cet espace-mémoire sera la valeur de `p`. Cette opération consistant à réserver un espace-mémoire pour stocker l'objet pointé s'appelle *allocation dynamique*. Elle se fait en C par la fonction `malloc` de la librairie standard `stdlib.h`.

Sa syntaxe est

`malloc(nombre-octets)`

Cette fonction retourne un pointeur de type `char *` pointant vers un objet de taille `nombre-octets` octets. Pour initialiser des pointeurs vers des objets qui ne sont pas de type `char`, il faut convertir le type de la sortie de la fonction `malloc` à l'aide d'un *cast*. L'argument `nombre-octets` est souvent donné à l'aide de la fonction `sizeof()` qui renvoie le nombre d'octets utilisés pour stocker un objet.

Ainsi, pour initialiser un pointeur vers un entier, on écrit :

```
#include <stdlib.h>
int *p;
p = (int*)malloc(sizeof(int));
```

On aurait pu écrire également :

```
p = (int*)malloc(4);
```

puisqu'un objet de type `int` est stocké sur 4 octets. Mais on préférera la première écriture qui a l'avantage d'être portable.

Le programme suivant

Exemple

```
#include <stdio.h>
#include <stdlib.h>
main()
{
    int i = 3 ;
    int *p ;
    printf("valeur de p avant initialisation = %ld\n",p) ;
    p = (int*)malloc(sizeof(int)) ;
    printf("valeur de p apres initialisation = %ld\n",p) ;
    *p = i ;
    printf("valeur de *p = %d\n",*p) ;
}
```

définit un pointeur `p` sur un objet `*p` de type `int`, et affecte à `*p` la valeur de la variable `i`. Il imprime à l'écran :

```
valeur de p avant initialisation = 0
valeur de p apres initialisation = 5368711424
valeur de *p = 3
```

Avant l'allocation dynamique, on se trouve dans la configuration :

objet	adresse	valeur
`i`	4831836000	3
`p`	4831836004	0

A ce stade, `*p` n'a aucun sens. En particulier, toute manipulation de la variable `*p` générerait une violation mémoire, détectable à l'exécution par le message d'erreur `Segmentation fault`.

L'allocation dynamique a pour résultat d'attribuer une valeur à `p` et de réserver à cette adresse un espace-mémoire composé de 4 octets pour stocker la valeur de `*p`. On a alors

objet	adresse	valeur
`i`	4831836000	3
`p`	4831836004	5368711424
`*p`	5368711424	? (int)

`*p` est maintenant définie mais sa valeur n'est pas initialisée. Cela signifie que `*p` peut valoir n'importe quel entier (celui qui se trouvait précédemment à cette adresse). L'affectation `*p = i ;` a enfin pour résultat d'affecter à `*p` la valeur de `i`. A la fin du programme, on a donc

objet	adresse	valeur
i	4831836000	3
p	4831836004	5368711424
*p	5368711424	3

Il est important de comparer le programme précédent avec

Exemple

```
main()
{
    int i = 3 ;
    int *p ;
```

```
    p = &i ;
}
```

qui correspond à la situation :

objet	adresse	valeur
i	4831836000	3
p	4831836004	4831836000
*p	4831836000	3

Dans ce dernier cas, les variables `i` et `*p` sont identiques (elles ont la même adresse) ce qui implique que toute modification de l'une modifie l'autre. Ceci n'était pas vrai dans l'exemple précédent où `*p` et `i` avaient la même valeur mais des adresses différentes.

On remarquera que le dernier programme ne nécessite pas d'allocation dynamique puisque l'espace-mémoire à l'adresse `&i` est déjà réservé pour un entier.

La fonction `malloc` permet également d'allouer un espace pour plusieurs objets contigus en mémoire. On peut écrire par exemple :

— Exemple —

```
#include <stdio.h>
#include <stdlib.h>
main()
{
    int i = 3 ;
    int j = 6 ;
    int *p ;
    p = (int*)malloc(2 * sizeof(int)) ;
    *p = i ;
    *(p + 1) = j ;
    printf("p = %ld \t *p = %d \t p+1 = %ld \t *(p+1) = %d \n",p,*p,p+1,*(p+1)) ;
}
```

On a ainsi réservé, à l'adresse donnée par la valeur de `p`, 8 octets en mémoire, qui permettent de stocker 2 objets de type `int`. Le programme affiche :
`p = 5368711424 *p = 3 p+1 = 5368711428 *(p+1) = 6 .`

La fonction `calloc` de la librairie `stdlib.h` a le même rôle que la fonction `malloc` mais elle initialise en plus l'objet pointé `*p` à zéro. Sa syntaxe est

`calloc(nb-objets,taille-objets)`

Ainsi, si `p` est de type `int*`, l'instruction :

`p = (int*)calloc(N,sizeof(int)) ;`

est strictement équivalente à :

```
p = (int*)malloc(N * sizeof(int)) ;
for (i = 0 ; i < N ; i++)
    *(p + i) = 0 ;
```

L'emploi de `calloc` est simplement plus rapide.

Enfin, lorsque l'on n'a plus besoin de l'espace-mémoire alloué dynamiquement (c'est-à-dire quand on n'utilise plus le pointeur `p`), il faut libérer cette place en mémoire. Ceci se fait à l'aide de l'instruction `free` qui a pour syntaxe :

`free(nom-du-pointeur) ;`

A toute instruction de type `malloc` ou `calloc` doit être associée une instruction de type `free`.

5 Pointeurs et tableaux

5.1 Pointeurs et tableaux à une dimension

Tout tableau en C est en fait un pointeur constant. Dans la déclaration

`int tab[10] ;`

`tab` est un pointeur constant (non modifiable) dont la valeur est l'adresse du premier élément du tableau. Autrement dit, `tab` a pour valeur `&tab[0]`. On peut donc utiliser un pointeur initialisé à `tab` pour parcourir les éléments du tableau.

— Exemple —

```
#define N 5
int tab[5] = {1, 2, 6, 0, 7};
main()
{
    int i;
    int *p;
    p = tab;
    for (i = 0; i < N; i++)
    {
       printf(" %d \n",*p);
       p++;
    }
}
```

On accède à l'élément d'indice `i` du tableau `tab` grâce à l'opérateur d'indexation `[]`, par l'expression `tab[i]`. Cet opérateur d'indexation peut en fait s'appliquer à tout objet `p` de type `pointeur`. Il est lié à l'opérateur d'indirection `*` par la formule :

`p[i] = *(p + i)`

Pointeurs et tableaux se manipulent donc exactement de même manière. Par exemple, le programme précédent peut aussi s'écrire

— Exemple —

```
#define N 5
int tab[5] = {1, 2, 6, 0, 7};
main()
{
    int i;
    int *p;
    p = tab;
    for (i = 0; i < N; i++)
       printf(" %d \n", p[i]);
}
```

Toutefois, la manipulation de tableaux, et non de pointeurs, possède certains inconvénients dûs au fait qu'un tableau est un pointeur constant. Ainsi, on ne peut pas créer de tableaux dont la taille est une variable du programme, on ne peut pas créer de tableaux bidimensionnels dont les lignes n'ont pas toutes

le même nombre d'éléments. Ces opérations deviennent possibles dès que l'on manipule des pointeurs alloués dynamiquement. Ainsi, pour créer un tableau d'entiers à n éléments où n est une variable du programme, on écrit :

— Exemple —

```
#include <stdlib.h>
main()
{
    int n ;
    int *tab ;

    ...
    tab = (int*)malloc(n * sizeof(int)) ;
    ...
    free(tab) ;
}
```

Si on veut en plus que tous les éléments du tableau `tab` soient initialisés à zéro, on remplace l'allocation dynamique avec `malloc` par :

```
tab = (int*)calloc(n, sizeof(int)) ;
```

Les éléments de `tab` sont manipulés avec l'opérateur d'indexation `[]`, exactement comme pour les tableaux.

Les deux différences principales entre un tableau et un pointeur sont :
- un pointeur doit toujours être initialisé, soit par une allocation dynamique, soit par affectation d'une expression adresse, par exemple `p = &i ;`
- un tableau n'est pas une *lvalue* ; il ne peut donc pas figurer à gauche d'un opérateur d'affectation. En particulier, un tableau ne supporte pas l'arithmétique (on ne peut pas écrire `tab++ ;`).

5.2 Pointeurs et tableaux à plusieurs dimensions

Un tableau à deux dimensions est, par définition, un tableau de tableaux. Il s'agit donc en fait d'un pointeur vers un pointeur. Considérons le tableau à deux dimensions défini par :

```
int tab[M][N] ;
```

`tab` est un pointeur, qui pointe vers un objet lui-même de type pointeur d'entier. `tab` a une valeur constante égale à l'adresse du premier élément du tableau, `&tab[0][0]`. De même `tab[i]`, pour `i` entre `0` et `M-1`, est un pointeur constant vers un objet de type entier, qui est le premier élément de la ligne d'indice `i`. `tab[i]` a donc une valeur constante qui est égale à `&tab[i][0]`.

Exactement comme pour les tableaux à une dimension, les pointeurs de pointeurs ont de nombreux avantages sur les tableaux multi-dimensionnés.

On déclare un pointeur qui pointe sur un objet de type type `*` (deux dimensions) de la même manière qu'un pointeur, c'est-à-dire

```
type **nom-du-pointeur ;
```

De même un pointeur qui pointe sur un objet de type type `**` (équivalent à un tableau à 3 dimensions) se déclare par :

```
type ***nom-du-pointeur ;
```

Par exemple, pour créer avec un pointeur de pointeur une matrice à k lignes et n colonnes à coefficients entiers, on écrit :

— Exemple —

```
main()
```

```
{
    int k, n ;
    int **tab ;

    tab = (int**)malloc(k * sizeof(int*)) ;
    for (i = 0 ; i < k ; i++)
       tab[i] = (int*)malloc(n * sizeof(int)) ;
       ....

    for (i = 0 ; i < k ; i++)
       free(tab[i]) ;
    free(tab) ;
}
```

La première allocation dynamique réserve pour l'objet pointé par `tab` l'espace-mémoire correspondant à `k` pointeurs sur des entiers. Ces `k` pointeurs correspondent aux lignes de la matrice. Les allocations dynamiques suivantes réservent pour chaque pointeur `tab[i]` l'espace-mémoire nécessaire pour stocker `n` entiers.

Si on désire en plus que tous les éléments du tableau soient initialisés à zéro, il suffit de remplacer l'allocation dynamique dans la boucle for par :

```
tab[i] = (int*)calloc(n, sizeof(int)) ;
```

Contrairement aux tableaux à deux dimensions, on peut choisir des tailles différentes pour chacune des lignes `tab[i]`.

Par exemple, si l'on veut que `tab[i]` contienne exactement $i + 1$ éléments, on écrit :

```
for (i = 0 ; i < k ; i++)
    tab[i] = (int*)malloc((i + 1) * sizeof(int)) ;
```

5.3 Exercice

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

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

```
#include <stdio.h>

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

#define DECALAGE 5

/*****************************************************************************/
/*                                                                           */
/*                                  crypt                                    */
/*                                                                           */
/*                 But :                                                     */
/*                  Crypte le caractère passé en paramètre                    */
/*                                                                           */
/*                              Interface :                                  */
```

```
/*                    p : pointe le caractère à crypter                    */
/*                                                                         */
/*************************************************************************/
void crypt(char *p)
{
enum {BAS, HAUT};
int casse;

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

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

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

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

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

return 0;
}
```

5.4 Passage de tableau en paramètre

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

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

— Exemple —

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

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

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

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

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

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

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

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

— Exemple —

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

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

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

— Exemple —

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

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

Malheureusement, il n'est pas possible d'omettre la taille du tableau dans les paramètres formels et effectifs de la fonction. En effet, le paramètre passé est une adresse, c'est-à-dire un entier, et non une adresse avec l'information qu'il s'agit d'un tableau avec sa longueur.

Par exemple, examinons le programme suivant :

— Exemple —

```
#include <stdio.h>

int som (int t[]) ; /* Déclaration d'une fonction qui recevra l'adresse d'un tableau.
                       En fait, on aurait aussi pu bien déclarer : int som (int * t) */
             /* t[] indique donc seulement qu'il s'agit d'un pointeur vers un entier */

int main () {
    int t1[3]={1,2,3}, t2[4]={4,5,6,7}, t3[5]={8,9,10,11,12} ;
    int s1, s2 ;
```

```
    s1 = som(t1) ;
    s2 = som(t2) + som(t3) ;
    printf("som(t1) à l'adresse %d = %d \n", t1, s1) ;
    printf("som(t2) à l'adresse %d = %d \n", t2, som(t2)) ;
    printf("som(t3) à l'adresse %d, de taille %d = %d \n", t3, sizeof(t3)/sizeof(int),
som(t3)) ;
    printf("som(%d) + som(%d) = %d \n", t2, t3, s2) ;
    return 0 ;
}

int som (int t[]) {
int s = 0, i ;
printf("Dans som : taille de t : %d \n", sizeof(&t)/sizeof(int)) ;
for (i=0 ; i < sizeof(&t)/sizeof(int) ; i++) {
s += t[i] ;
}
return(s) ;
}
```

L'exécution de ce programme produira :

```
[Session started at 2007-05-17 12 :48 :59 +0200.]
Dans som : taille de t : 1
Dans som : taille de t : 1
Dans som : taille de t : 1
som(t1) à l'adresse -1073743780 = 1
Dans som : taille de t : 1
som(t2) à l'adresse -1073743796 = 4
Dans som : taille de t : 1
som(t3) à l'adresse -1073743816, de taille 5 = 8
som(-1073743796) + som(-1073743816) = 12

Test-C-pointeurs has exited with status 0.
```

L'information de la taille de tableau n'est donc pas transmise avec le paramètre d'appel de la fonction `som`.

Remarque :

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

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

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

5.5 Pointeurs et chaînes de caractères

On a vu précédemment qu'une chaîne de caractères était un tableau à une dimension d'objets de type `char`, se terminant par le caractère nul `'\0'`. On peut donc manipuler toute chaîne de caractères à l'aide d'un pointeur sur un objet de type `char`. On peut faire subir à une chaîne définie par :

```
char *chaine ;
```

des affectations comme :

```
chaine = "ceci est une chaine";
```

et toute opération valide sur les pointeurs, comme l'instruction `chaine++;`. Ainsi, le programme suivant imprime le nombre de caractères d'une chaîne (sans compter le caractère nul).

Exemple

```
#include <stdio.h>
main()
{
int i;
char *chaine;

    chaine = "chaine de caracteres";
    for (i = 0; *chaine != '\0'; i++)
       chaine++;
    printf("nombre de caracteres = %d\n",i);
}
```

La fonction donnant la longueur d'une chaîne de caractères, définie dans la librairie standard `string.h`, procède de manière identique. Il s'agit de la fonction `strlen` dont la syntaxe est

```
strlen(chaine);
```

où `chaine` est un pointeur sur un objet de type `char`. Cette fonction renvoie un entier dont la valeur est égale à la longueur de la chaîne passée en argument (moins le caractère `'\0'`). L'utilisation de pointeurs de caractère et non de tableaux permet par exemple de créer une chaîne correspondant à la concaténation de deux chaînes de caractères :

Exemple

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
main()  {
    int i;
    char *chaine1, *chaine2, *res, *p;

    chaine1 = "chaine ";
    chaine2 = "de caracteres";
    res = (char*)malloc((strlen(chaine1) + strlen(chaine2)) * sizeof(char));
    p = res;
    for (i = 0; i < strlen(chaine1); i++)
       *p++ = chaine1[i];
    for (i = 0; i < strlen(chaine2); i++)
       *p++ = chaine2[i];
    printf("%s\n",res);
}
```

On remarquera l'utilisation d'un pointeur intermédiaire `p` qui est indispensable dès que l'on fait des opérations de type incrémentation. En effet, si on avait incrémenté directement la valeur de `res`, on aurait évidemment "perdu" la référence sur le premier caractère de la chaîne. Par exemple,

Exemple

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

```
main()
{
    int i ;
    char *chaine1, *chaine2, *res ;

    chaine1 = "chaine " ;
    chaine2 = "de caracteres" ;
    res = (char*)malloc((strlen(chaine1) + strlen(chaine2)) * sizeof(char)) ;
    for (i = 0 ; i < strlen(chaine1) ; i++)
       *res++ = chaine1[i] ;
    for (i = 0 ; i < strlen(chaine2) ; i++)
       *res++ = chaine2[i] ;
    printf("\n nombre de caracteres de res = %d\n",strlen(res)) ;
}
```

imprime la valeur `0`, puisque `res` a été modifié au cours du programme et pointe maintenant sur le caractère nul.

6 Pointeurs et structures

6.1 Pointeurs sur une structure

Contrairement aux tableaux, les objets de type structure en C sont des *lvalues*. Ils possèdent une adresse, correspondant à l'adresse du premier élément du premier membre de la structure. On peut donc manipuler des pointeurs sur des structures. Ainsi, le programme suivant crée, à l'aide d'un pointeur, un tableau d'objets de type `structure`.

— Exemple —

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

struct eleve
{
    char nom[20] ;
    int date ;
} ;

typedef struct eleve *classe ;

main()
{
    int n, i ;
    classe tab ;

    printf("nombre d'eleves de la classe = ") ;
    scanf("%d",&n) ;
    tab = (classe)malloc(n * sizeof(struct eleve)) ;

    for (i =0 ; i < n ; i++)
    {
       printf("\n saisie de l'eleve numero %d \n",i) ;
       printf("nom de l'eleve = ") ;
       scanf("%s",&tab[i].nom) ;
       printf("\n date de naissance JJMMAA = ") ;
       scanf("%d",&tab[i].date) ;
    }
    printf("\n Entrez un numero ") ;
    scanf("%d",&i) ;
```

```
    printf("\n Eleve numero %d :",i) ;
    printf("\n nom = %s",tab[i].nom) ;
    printf("\n date de naissance = %d \n",tab[i].date) ;
    free(tab) ;
}
```

Si `p` est un pointeur sur une structure, on peut accéder à un membre de la structure pointé par l'expression `(*p).membre`.

L'usage de parenthèses est ici indispensable car l'opérateur d'indirection `*` à une priorité plus élevée que l'opérateur de membre de structure. Cette notation peut être simplifiée grâce à l'opérateur pointeur de membre de structure, noté `->`. L'expression précédente est strictement équivalente à :

```
p->membre
```

Ainsi, dans le programme précédent, on peut remplacer `tab[i].nom` et `tab[i].date` respectivement par `(tab + i)->nom` et `(tab + i)->date`.

6.2 Structures auto-référencées

On a souvent besoin en C de modèles de structure dont un des membres est un pointeur vers une structure de même modèle. Cette représentation permet en particulier de construire des listes chaînées. En effet, il est possible de représenter une liste d'éléments de même type par un tableau (ou un pointeur). Toutefois, cette représentation, dite *contiguë*, impose que la taille maximale de la liste soit connue a priori (on a besoin du nombre d'éléments du tableau lors de l'allocation dynamique). Pour résoudre ce problème, on utilise une représentation chaînée : l'élément de base de la chaîne est une structure appelée *cellule* qui contient la valeur d'un élément de la liste et un pointeur sur l'élément suivant. Le dernier élément pointe sur la liste vide `NULL`. La liste est alors définie comme un pointeur sur le premier élément de la chaîne.

Pour représenter une liste d'entiers sous forme chaînée, on crée le modèle de structure `cellule` qui a deux champs : un champ `valeur` de type `int`, et un champ `suivant` de type `pointeur` sur une `struct cellule`. Une liste sera alors un objet de type pointeur sur une `struct cellule`. Grâce au mot-clef `typedef`, on peut définir le type `liste`, synonyme du type pointeur sur une `struct cellule`.

— Exemple —

```
struct cellule
{
    int valeur ;
    struct cellule *suivant ;
} ;

typedef struct cellule *liste ;
```

Un des avantages de la représentation chaînée est qu'il est très facile d'insérer un élément à un endroit quelconque de la liste. Ainsi, pour insérer un élément en tête de liste, on utilise la fonction suivante :

— Exemple —

```
liste insere(int element, liste Q)
{
    liste L ;
    L = (liste)malloc(sizeof(struct cellule)) ;
    L->valeur = element ;
    L->suivant = Q ;
    return(L) ;
}
```

Le programme suivant crée une liste d'entiers et l'imprime à l'écran :

Exemple

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

struct cellule
{
    int valeur ;
    struct cellule *suivant ;
} ;

typedef struct cellule *liste ;

liste insere(int element, liste Q)
{
    liste L ;
    L = (liste)malloc(sizeof(struct cellule)) ;
    L->valeur = element ;
    L->suivant = Q ;
    return(L) ;
}

main()
{
    liste L, P ;

    L = insere(1,insere(2,insere(3,insere(4,NULL)))) ;
    printf("\n impression de la liste :\n") ;
    P = L ;
    while (P != NULL)
    {
       printf("%d \t",P->valeur) ;
       P = P->suivant ;
    }
}
```

On utilisera également une structure auto-référencée pour créer un arbre binaire :

```
struct noeud
{
int valeur ;
struct noeud *fils_gauche ;
struct noeud *fils_droit ;
} ;

typedef struct noeud *arbre ;
```

Chapitre 5

Flots de données et gestion des fichiers

Sommaire

Le C offre la possibilité de lire et d'écrire des données dans un fichier.

Pour des raisons d'efficacité, les accès à un fichier se font par l'intermédiaire d'une mémoire-tampon (*buffer*), ce qui permet de réduire le nombre d'accès aux périphériques (disque...).

Pour pouvoir manipuler un fichier, un programme a besoin d'un certain nombre d'informations : l'adresse de l'endroit de la mémoire-tampon où se trouve le fichier, la position de la tête de lecture, le mode d'accès au fichier (lecture ou écriture) ... Ces informations sont rassemblées dans une structure dont le type, `FILE *`, est défini dans `stdio.h`. Un objet de type `FILE *` est appelé *flot de données* (en anglais, *stream*).

Avant de lire ou d'écrire dans un fichier, on notifie son accès par la commande `fopen`. Cette fonction prend comme argument le nom du fichier, négocie avec le système d'exploitation et initialise un flot de données, qui sera ensuite utilisé lors de l'écriture ou de la lecture. Après les traitements, on annule la liaison entre le fichier et le flot de données grâce à la fonction `fclose`.

1 Ouverture et fermeture d'un fichier

1.1 La fonction `fopen`

Cette fonction, de type `FILE*` ouvre un fichier et lui associe un flot de données. Sa syntaxe est :

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

La valeur retournée par `fopen` est un flot de données. Si l'exécution de cette fonction ne se déroule pas normalement, la valeur retournée est le pointeur `NULL`. Il est donc recommandé de toujours tester si la valeur renvoyée par la fonction fopen est égale à `NULL` afin de détecter les erreurs (lecture d'un fichier inexistant...).

Le premier argument de `fopen` est le nom du fichier concerné, fourni sous forme d'une chaîne de caractères. On préférera définir le nom du fichier par une constante symbolique au moyen de la directive `#define` plutôt que d'expliciter le nom de fichier dans le corps du programme.

Le second argument, `mode`, est une chaîne de caractères qui spécifie le mode d'accès au fichier. Les spécificateurs de mode d'accès diffèrent suivant le type de fichier considéré. On distingue :

- les *fichiers textes*, pour lesquels les caractères de contrôle (retour à la ligne ...) seront interprétés en tant que tels lors de la lecture et de l'écriture ;
- les *fichiers binaires*, pour lesquels les caractères de contrôle se sont pas interprétés.

Les différents modes d'accès sont les suivants :

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

Ces modes d'accès ont pour particularités :

- Si le mode contient la lettre `r`, le fichier doit exister.
- Si le mode contient la lettre `w`, le fichier peut ne pas exister. Dans ce cas, il sera créé. Si le fichier existe déjà, son ancien contenu sera perdu.
- Si le mode contient la lettre `a`, le fichier peut ne pas exister. Dans ce cas, il sera créé. Si le fichier existe déjà, les nouvelles données seront ajoutées à la fin du fichier précédent.

Trois flots standard peuvent être utilisés en C sans qu'il soit nécessaire de les ouvrir ou de les fermer :

- `stdin` (standard input) : unité d'entrée (par défaut, le clavier) ;
- `stdout` (standard output) : unité de sortie (par défaut, l'écran) ;
- `stderr` (standard error) : unité d'affichage des messages d'erreur (par défaut, l'écran).

Il est fortement conseillé d'afficher systématiquement les messages d'erreur sur stderr afin que ces messages apparaissent à l'écran même lorsque la sortie standard est redirigée.

1.2 La fonction `fclose`

Elle permet de fermer le flot qui a été associé à un fichier par la fonction fopen. Sa syntaxe est :

```
fclose(flot)
```

où `flot` est le flot de type `FILE*` retourné par la fonction `fopen` correspondant.

La fonction `fclose` retourne un entier qui vaut zéro si l'opération s'est déroulée normalement (et une valeur non nulle en cas d'erreur).

2 Les entrées-sorties formatées

2.1 La fonction fprintf

La fonction `fprintf`, analogue à `printf`, permet d'écrire des données dans un fichier. Sa syntaxe est :

```
fprintf(flot,"chaîne de contrôle",expression-1, ..., expression-n)
```

où `flot` est le flot de données retourné par la fonction `fopen`. Les spécifications de format utilisées pour la fonction `fprintf` sont les mêmes que pour `printf`.

2.2 La fonction fscanf

La fonction `fscanf`, analogue à `scanf`, permet de lire des données dans un fichier. Sa syntaxe est semblable à celle de `scanf` :

```
fscanf(flot,"chaîne de contrôle",argument-1,...,argument-n)
```

où `flot` est le flot de données retourné par `fopen`. Les spécifications de format sont ici les mêmes que celles de la fonction `scanf`.

3 Impression et lecture de caractères

Similaires aux fonctions `getchar` et `putchar`, les fonctions `fgetc` et `fputc` permettent respectivement de lire et d'écrire un caractère dans un fichier. La fonction `fgetc`, de type `int`, retourne le caractère lu dans le fichier. Elle retourne la constante `EOF` lorsqu'elle détecte la fin du fichier. Son prototype est :

```
int fgetc(FILE* flot) ;
```

où `flot` est le flot de type `FILE*` retourné par la fonction `fopen`. Comme pour la fonction `getchar`, il est conseillé de déclarer de type `int` la variable destinée à recevoir la valeur de retour de `fgetc` pour pouvoir détecter correctement la fin de fichier.

La fonction `fputc` écrit `caractere` dans le flot de données :

```
int fputc(int caractere, FILE *flot)
```

Elle retourne l'entier correspondant au caractère lu (ou la constante `EOF` en cas d'erreur).

Il existe également deux versions optimisées des fonctions `fgetc` et `fputc` qui sont implémentées par des macros. Il s'agit respectivement de `getc` et `putc`. Leur syntaxe est similaire à celle de `fgetc` et `fputc` :

```
int getc(FILE* flot) ;
int putc(int caractere, FILE *flot)
```

Ainsi, le programme suivant lit le contenu du fichier texte entree, et le recopie caractère par caractère dans le fichier sortie :

— Exemple —

```
#include <stdio.h>
#include <stdlib.h>
#define ENTREE "entree.txt"
#define SORTIE "sortie.txt"

int main(void)
{
    FILE *f_in, *f_out ;
    int c ;

    if ((f_in = fopen(ENTREE,"r")) == NULL)
       {
```

```
        fprintf(stderr, "\nErreur : Impossible de lire le fichier %s\n",ENTREE) ;
        return(EXIT_FAILURE) ;
        }
    if ((f_out = fopen(SORTIE,"w")) == NULL)
        {
        fprintf(stderr, "\nErreur : Impossible d'ecrire dans le fichier %s\n", SORTIE) ;
        return(EXIT_FAILURE) ;
        }
    while ((c = fgetc(f_in)) != EOF)
        fputc(c, f_out) ;
    fclose(f_in) ;
    fclose(f_out) ;
    return(EXIT_SUCCESS) ;
}
```

4 Relecture d'un caractère

Il est possible de replacer un caractère dans un flot au moyen de la fonction `ungetc` :

`int ungetc(int caractere, FILE *flot) ;`

Cette fonction place le caractère `caractere` (converti en `unsigned char`) dans le flot `flot`. En particulier, si `caractere` est égal au dernier caractère lu dans le flot, elle annule le déplacement provoqué par la lecture précédente. Toutefois, `ungetc` peut être utilisée avec n'importe quel caractère (sauf `EOF`). Par exemple, l'exécution du programme suivant :

Exemple

```
#include <stdio.h>
#include <stdlib.h>
#define ENTREE "entree.txt"

int main(void)
{
    FILE *f_in ;
    int c ;

    if ((f_in = fopen(ENTREE,"r")) == NULL)
        {
        fprintf(stderr, "\nErreur : Impossible de lire le fichier %s\n",ENTREE) ;
        return(EXIT_FAILURE) ;
        }

    while ((c = fgetc(f_in)) != EOF)
        {
        if (c == '0')
            ungetc('.',f_in) ;
        putchar(c) ;
        }
    fclose(f_in) ;
    return(EXIT_SUCCESS) ;
}
```

sur le fichier `entree.txt` dont le contenu est `097023` affiche à l'écran `0.970.23`

5 Les entrées-sorties binaires

Les fonctions d'entrées-sorties binaires permettent de transférer des données dans un fichier sans transcodage. Elles sont donc plus efficaces que les fonctions d'entrée-sortie standard, mais les fichiers produits ne sont pas portables puisque le codage des données dépend des machines.

Elles sont notamment utiles pour manipuler des données de grande taille ou ayant un type composé. Leurs prototypes sont :

```
size_t fread(void *pointeur, size_t taille, size_t nombre, FILE *flot) ;
size_t fwrite(void *pointeur, size_t taille, size_t nombre, FILE *flot) ;
```

où `pointeur` est l'adresse du début des données à transférer, `taille` la taille des objets à transférer, `nombre` leur nombre. Rappelons que le type `size_t`, défini dans `stddef.h`, correspond au type du résultat de l'évaluation de `sizeof`. Il s'agit du plus grand type entier non signé.

La fonction `fread` lit les données sur le flot `flot` et la fonction `fwrite` les écrit. Elles retournent toutes deux le nombre de données transférées.

Par exemple, le programme suivant écrit un tableau d'entiers (contenant les 50 premiers entiers) avec `fwrite` dans le fichier sortie, puis lit ce fichier avec `fread` et imprime les éléments du tableau.

Exemple

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

#define NB 50
#define F_SORTIE "sortie"

int main(void)
{
    FILE *f_in, *f_out ;
    int *tab1, *tab2 ;
    int i ;

    tab1 = (int*)malloc(NB * sizeof(int)) ;
    tab2 = (int*)malloc(NB * sizeof(int)) ;
    for (i = 0 ; i < NB ; i++)
       tab1[i] = i ;

    /* ecriture du tableau dans F_SORTIE */
    if ((f_out = fopen(F_SORTIE, "w")) == NULL)
       {
       fprintf(stderr, "\nImpossible d'ecrire dans le fichier %s\n",F_SORTIE) ;
       return(EXIT_FAILURE) ;
       }
    fwrite(tab1, NB * sizeof(int), 1, f_out) ;
    fclose(f_out) ;

    /* lecture dans F_SORTIE */
    if ((f_in = fopen(F_SORTIE, "r")) == NULL)
{
       fprintf(stderr, "\nImpossible de lire dans le fichier %s\n",F_SORTIE) ;
       return(EXIT_FAILURE) ;
}
    fread(tab2, NB * sizeof(int), 1, f_in) ;
    fclose(f_in) ;
    for (i = 0 ; i < NB ; i++)
       printf("%d\t",tab2[i]) ;
    printf("\n") ;
    return(EXIT_SUCCESS) ;
}
```

Les éléments du tableau sont bien affichés à l'écran. Par contre, on constate que le contenu du fichier `sortie` n'est pas encodé.

6 Positionnement dans un fichier

Les différentes fonctions d'entrées-sorties permettent d'accéder à un fichier en *mode séquentiel* : les données du fichier sont lues ou écrites les unes à la suite des autres. Il est également possible d'accéder à un fichier en *mode direct*, c'est-à-dire que l'on peut se positionner à n'importe quel endroit du fichier. La fonction `fseek` permet de se positionner à un endroit précis ; elle a pour prototype :

```
int fseek(FILE *flot, long deplacement, int origine) ;
```

La variable `deplacement` détermine la nouvelle position dans le fichier. Il s'agit d'un déplacement relatif par rapport à l'origine ; il est compté en nombre d'octets. La variable origine peut prendre trois valeurs :

- `SEEK_SET` (égale à 0) : début du fichier ;
- `SEEK_CUR` (égale à 1) : position courante ;
- `SEEK_END` (égale à 2) : fin du fichier.

La fonction

```
int rewind(FILE *flot) ;
```

permet de se positionner au début du fichier. Elle est équivalente à :

```
fseek(flot, 0, SEEK_SET) ;
```

La fonction :

```
long ftell(FILE *flot) ;
```

retourne la position courante dans le fichier (en nombre d'octets depuis l'origine).

Par exemple :

Exemple

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

#define NB 50
#define F_SORTIE "sortie"

int main(void)
{
    FILE *f_in, *f_out ;
    int *tab ;
    int i ;

    tab = (int*)malloc(NB * sizeof(int)) ;
    for (i = 0 ; i < NB ; i++)
       tab[i] = i ;

    /* ecriture du tableau dans F_SORTIE */
    if ((f_out = fopen(F_SORTIE, "w")) == NULL)
       {
       fprintf(stderr, "\nImpossible d'ecrire dans le fichier %s\n",F_SORTIE) ;
       return(EXIT_FAILURE) ;
       }
    fwrite(tab, NB * sizeof(int), 1, f_out) ;
    fclose(f_out) ;

    /* lecture dans F_SORTIE */
    if ((f_in = fopen(F_SORTIE, "r")) == NULL)
       {
```

```
        fprintf(stderr, "\nImpossible de lire dans le fichier %s\n",F_SORTIE) ;
        return(EXIT_FAILURE) ;
        }

    /* on se positionne a la fin du fichier */
    fseek(f_in, 0, SEEK_END) ;
    printf("\n position %ld", ftell(f_in)) ;
    /* deplacement de 10 int en arriere */
    fseek(f_in, -10 * sizeof(int), SEEK_END) ;
    printf("\n position %ld", ftell(f_in)) ;
    fread(&i, sizeof(i), 1, f_in) ;
    printf("\t i = %d", i) ;
    /* retour au debut du fichier */
    rewind(f_in) ;
    printf("\n position %ld", ftell(f_in)) ;
    fread(&i, sizeof(i), 1, f_in) ;
    printf("\t i = %d", i) ;
    /* deplacement de 5 int en avant */
    fseek(f_in, 5 * sizeof(int), SEEK_CUR) ;
    printf("\n position %ld", ftell(f_in)) ;
    fread(&i, sizeof(i), 1, f_in) ;
    printf("\t i = %d\n", i) ;
    fclose(f_in) ;
    return(EXIT_SUCCESS) ;
}
```

L'exécution de ce programme affiche à l'écran :

```
position 200
position 160   i = 40
position 0     i = 0
position 24    i = 6
```

On constate en particulier que l'emploi de la fonction `fread` provoque un déplacement correspondant à la taille de l'objet lu à partir de la position courante.

Index