Langages et Concepts de Programmation
Structures de données et algorithmes en C

Cours 1A 2012-2013

Jean-Jacques Girardot, Marc Roelens
girardot@emse.fr, roelens@emse.fr
Octobre 2012

École Nationale Supérieure des Mines de Saint-Etienne
158 Cours Fauriel
42023 SAINT-ÉTIENNE CEDEX

Version du 4 octobre 2012.

Première partie

Cours

Introduction

Avant-Propos

Ce cours [2] fait suite au cours d'Introduction à l'Informatique [1]. Comme le précédent, ce cours se veut aussi bien une continuation de l'apprentissage de l'algorithmique et de la programmation, qu'une poursuite de l'étude du langage `C`.

En parallèle à ce cours, les élèves ont à réaliser un mini-projet de programmation, destiné à mettre en œuvre de façon pratique les concepts et techniques abordées dans le cours.

Dans le cadre du pôle de modélisation mathématique, les élèves suivent en parallèle le cours de Recherche opérationnelle. Ce cours se termine par deux séances de travaux pratiques, consistant à implémenter grâce aux compétences acquises dans le présent cours un algorithme de résolution d'un problème « classique » de recherche opérationnelle.

Déroulement du cours

Séances 1 et 2

But Savoir écrire des programmes accédant à leurs données en-dehors du programme lui-même.

Théorie La représentation externe des données. Entrées-sorties en `C`. Fichiers texte et fichiers binaires.

Pratique Lire des données simples (entiers, flottants, chaînes de caractères), des tableaux.

Formater des données en sortie, lire des données formatées en entrée.

Séances 3 et 4

But Savoir manipuler des structures de données du langage `C`.

Théorie Notion d'enregistrement (regroupement de données de types différents), champs d'un enregistrement. Structures de données comme paramètres ou résultat de procédures.

Pratique Construire une structure de données adaptée à un problème posé. Spécifier les procédures de manipulation de cette structure de données. Implémenter ces procédures.

Séances 5, 6 et 7

But Initiation au traitement des listes

Théorie Algorithmes sur listes. Allocation dynamique de structures de données, adresses (pointeurs).

Pratique Programmes de recherche en liste, d'interclassement, de tri de listes.

Chapitre 1

Entrées-sorties en C

1.1 Cours

Ce cours va nous permettre d'aborder les *entrées-sorties*, qui sont les opérations permettant à nos programmes de communiquer avec le monde extérieur.

Dans ces opérations, nous nous intéresserons entre autres à celles qui permettent de lire et d'écrire des données sur les *fichiers*.

Enfin, nous profiterons de l'occasion pour introduire le traitement des erreurs à l'exécution d'un programme au travers de la variable globale `errno` et de la procédure de bibliothèque `perror`.

1.1.1 Introduction

Les données que nous avons manipulées jusqu'à présent résidaient uniquement en mémoire centrale. Créées par le programme en cours d'exécution, elles étaient traitées, éventuellement imprimées, puis disparaissaient lorsque le programme se terminait. Certaines données informatiques, une fois créées, doivent au contraire être conservées pour une utilisation ultérieure : un jeu se souvient des *high scores* et des noms de joueurs, les données comptables d'une entreprise sont préservées pendant des années, les pages du web sont gardées pour pouvoir être fournies à la demande, etc.

Nous avons déjà décrit brièvement la structure des systèmes de fichiers, et la désignation des fichiers eux-mêmes. Nous avons utilisé jusqu'à présent ces fichiers pour la seule représentation des programmes, qu'ils soient en source, objet, ou exécutables. Nous allons aborder maintenant l'utilisation des fichiers pour la représentation et la conservation des données manipulées par les programmes.

1.1.2 Représentation des données dans un fichier

Il faut bien distinguer à ce niveau deux représentations différentes des données de nos programmes :

- ◦ la représentation *interne*, c'est-à-dire la forme sous laquelle ces données sont manipulées par le programme lui-même ; cette représentation est donc totalement dépendante du langage de programmation ;
- ◦ la représentation *externe*, c'est-à-dire le codage de ces données dans le but de les conserver dans un fichier ; cette représentation peut dépendre des moyens fournis par le système d'exploitation.

Nous avons vu, par exemple, que les données de nos programmes étaient accessibles sous forme de variables, ayant chacune un nom et un type. Ces déclarations de variables sont partie intégrante du langage de programmation, et le programmeur n'a donc que peu de choix (divers formats d'entiers et de nombres flottants, noms des variables).

A contrario, le langage C ne connaît pas la notion de fichier (ce n'est pas une notion propre au langage) ; les données stockées dans un fichier externe n'ont aucun format imposé par le langage lui-même : **il est de la responsabilité totale du programmeur de préciser sous quel(s) format(s) ses données sont représentées dans les fichiers externes**. Ces données utiles pour un programme peuvent ainsi être représentées, sous une forme éventuellement différente (compactée, codée...) de celle qui est utilisée par le programme. C'est toujours au programmeur de garantir que l'utilisation de ces données se fait à bon escient !

On rappelle qu'au niveau du système d'exploitation, un fichier est une simple suite d'octets, en général physiquement enregistrés sur des supports magnétiques, opto-électroniques, etc. Ces fichiers ont un *nom externe* (et une « position » : les *répertoires*), des *propriétés* (ou *attributs* : peut-on le lire, le modifier ? quand a-t-il été créé, etc) et un contenu, les données elles-mêmes.

Sous cette forme, les données ne sont pas directement accessibles par le processeur. Le système d'exploitation permet, grâce aux primitives d'entrées-sorties, d'avoir accès à un fichier, d'en lire et éventuellement d'en modifier le contenu, et de libérer l'accès à ce fichier, afin qu'il puisse être utilisé par d'autres programmes.

1.1.3 Bibliothèque standard des entrées-sorties

Ainsi que cela a été indiqué, le langage C ne contient pas la notion de fichier (rappelons que le langage C est un langage simple, voire simpliste... mais néanmoins extrêmement puissant).

Cependant, comme ce sont des objets incontournables pour bien des programmes, on a défini pour ce langage des bibliothèques de structures de données et de procédures permettant de réaliser les procédures d'entrées-sorties : cette bibliothèque, standardisée, est souvent appelée *bibliothèque standard des entrées-sorties* ou encore *STanDard Input-Output library*, d'où le nom du fichier d'en-tête contenant les structures de données et prototypes des procédures, à savoir le fichier `stdio.h` déjà maintes fois signalé.

> Il serait possible à tout un chacun de ne pas vouloir utiliser cette bibliothèque standard des entrées-sorties, mais on perdrait alors tout avantage de la standardisation : universalité, performance éprouvée...

Remarque : l'un des avantages de cette bibliothèque est aussi le fait que le langage C s'affranchit presque totalement des problèmes de portabilité entre systèmes d'exploitation différents

(Windows et Unix, notamment). On trouvera ainsi des versions spécialisées de la bibliothèque sur chaque environnement utilisé.

1.1.4 Définition des flots

Les mécanismes d'entrées-sorties en C font appel à la notion de *flot*. Un flot est une séquence d'octets, accessibles séquentiellement, qui peut être lue et/ou écrite par le programme. Les programmes C manipulent en particulier :

- un flot d'entrée, dit *entrée standard*, accessible par l'intermédiaire d'un objet nommé `stdin` ;
- un flot de sortie, dit *sortie standard*, accessible par l'intermédiaire d'un objet nommé `stdout` ;
- un flot de gestion des erreurs, accessible par l'intermédiaire d'un objet nommé `stderr`.

Ces trois flots sont habituellement associés au terminal (clavier, écran) de l'utilisateur. Par exemple, la procédure `printf` déjà présentée réalise toutes ses sorties sur le flot `stdout`.

Les objets `stdin`, `stdout` et `stderr` sont déclarés dans le fichier d'inclusion `stdio.h`, que nous connaissons bien. Les déclarations précises de ces objets sont :

```
extern FILE *stdin;
extern FILE *stdout;
extern FILE *stderr;
```

Le mot-clef `extern` indique que ces données ne sont pas définies dans le programme source lui-même, mais dans un module externe (la bibliothèque standard du C), qui sera inclus lors de la phase d'édition des liens.

Les objets sont de type `FILE *`, ce qui veut dire qu'ils sont des adresses (l'étoile) de structures spécifiques (de nom `FILE`) décrivant les caractéristiques des flots correspondants. Nous n'avons pas besoin de savoir précisément ce que contiennent ces structures, car le seul traitement que nous leur appliquerons sera de les utiliser comme paramètres de procédures d'entrées-sorties qui, elles, connaissent lesdites structures. Retenons simplement que, comme toute adresse, l'adresse `0` définie dans `stdio.h` par la macro-définition `NULL`, représente une adresse incorrecte, et sera toujours utilisée lorsqu'une procédure de manipulation de fichier voudra indiquer qu'elle a échoué.

Outre ces flots « standard », nous allons voir que des mécanismes du système permettent d'accéder, sous la forme de flots, au contenu des fichiers gérés par le système d'exploitation.

1.1.5 Flots et fichiers

1.1.5.1 Création d'un flot à partir d'un fichier

La construction d'un flot à partir d'un fichier externe se fait par la procédure `fopen(3)` [1]. On notera que dans la suite, on parlera d'ouverture de fichier pour désigner la construction d'un

1. Cette notation particulière indique que la procédure `fopen` fait partie de la section `3` du manuel en ligne : la section 3 contient en fait toutes les bibliothèques de procédures, et en particulier les opérations d'entrées-sorties du C. On peut consulter la documentation, sous Linux, par la commande `man -s 3 fopen`

flot à partir d'un fichier (le nom `fopen` dérivant de *file open*, soit *ouverture de fichier*).

Le prototype de cette procédure `fopen`, défini dans le traditionnel fichier `stdio.h`, est :

```
FILE *fopen (char nom[],char mode[]);
```

Le résultat de cette procédure `fopen` est l'adresse d'un objet de type `FILE` qui désigne le flot créé. En cas d'erreur, le résultat est l'adresse `0`, ou `NULL`.

Le premier paramètre `nom` indique le nom externe du fichier sous forme de chaîne de caractères, qui peut représenter un nom absolu comme `/etc/passwd`, ou relatif comme `toto.data` ou `../toto.out`. Le second paramètre indique la nature du flot que l'on veut construire ; c'est également une chaîne de caractères, qui peut prendre les valeurs et significations suivantes :

- `"r"` le flot est ouvert en lecture seulement ; il y a erreur si le fichier n'existe pas ou n'est pas accessible pour l'utilisateur ;
- `"w"` le flot est ouvert en écriture seulement ; si le fichier existe déjà, son contenu est préalablement détruit ; le fichier est créé s'il n'existe pas déjà ; il y a erreur si le fichier ne peut être créé ou ne peut être modifié par l'utilisateur ;
- `"a"` le flot est ouvert en écriture seulement, mais le contenu est conservé ; le fichier est crée s'il n'existe pas déjà ; toutes les écritures sont faites en fin de fichier (le `a` signifie *append*) ; il y a erreur si le fichier ne peut être créé ou ne peut être modifié par l'utilisateur ;
- `"r+"` le flot est ouvert en lecture et écriture, son contenu n'est pas modifié ; il y a erreur si le fichier n'existe pas, n'est pas accessible ou n'est pas modifiable ;
- `"w+"` le flot est ouvert en lecture et écriture, son contenu est préalablement détruit ; le fichier est créé s'il n'existe pas préalablement ; il y a erreur si le fichier n'est pas accessible, n'est pas modifiable ou ne peut être créé ;
- `"a+"` le flot est ouvert en lecture et écriture, son contenu initial est conservé, toutes les écritures sont faites à la fin du fichier ; il y a erreur si le fichier n'est pas accessible, ou ne peut être créé ;

1.1.5.2 Destruction d'un flot

Lorsque le programme n'a plus besoin d'un flot, il peut en demander la destruction explicite par la procédure :

```
int fclose (FILE *pf);
```

le résultat de la procédure étant 0 si tout se passe bien, ou -1 (encore défini dans `stdio.h` par la macro-définition `EOF` pour *End Of File*) en cas de problème.

Remarque 1 : la destruction d'un flot n'implique pas celle du fichier associé, mais simplement la libération des structures de données du programme permettant d'accéder à cette ressource ;

Remarque 2 : tout flot nécessite des ressources systèmes pour accéder à un fichier, et ces ressources sont en nombre limité. On a donc tout intérêt à fermer explicitement tout flot devenu inutile !

Remarque 3 : bien que non construits explicitement (par le programmeur) par des appels à `fopen`, les flots prédéfinis `stdin`, `stdout` et `stderr` peuvent également être fermés s'ils ne sont pas utiles grâce à la procédure `fclose`.

Remarque 4 : lors de la fin de vie normale du programme (en particulier en cas d'appel de la procédure `exit`), ou d'une fin de vie anormale (*plantage* du programme), les flots sont implicitement détruits. Cependant, il est de bonne pratique de les détruire explicitement s'ils ne sont plus utiles.

1.1.5.3 Réouverture d'un flot

On dispose d'un autre appel de re-création de flot :

```
FILE *freopen (char nom[],char mode[],FILE *old);
```

Lors de cet appel, le flot désigné par `old` est préalablement fermé, et le fichier désigné par `nom` est ouvert selon le mode `mode` en lieu et place de l'ancien flot `old`. Comme `fopen`, cette procédure rend la valeur `NULL` en cas d'erreur.

Voici un exemple d'utilisation de cette procédure :

```
if (freopen("toto.out","w",stdout)==NULL) {
  perror("toto.out");
  exit(2);
}
printf("Hello world\n");
...
```

L'appel à `freopen` ré-ouvre le flot `stdout` en écriture sur le fichier `toto.out`, et l'appel ultérieur à la procédure `printf` écrira donc le message dans le fichier `toto.out` et non pas sur l'écran.

Remarque importante : l'exemple ci-dessus donne la seule bonne façon de modifier les flots prédéfinis ; l'écriture

```
stdout=fopen("toto.out","w");
```

est totalement à bannir, et certains environnements la refusent carrément (pour des raisons trop complexes pour être détaillées ici).

Enfin, pour le coin de la culture, il existe également une procédure `fdopen`, souvent documentée dans la même page de manuel que `fopen` et `freopen`, dont le fonctionnement dépasse encore le niveau de ce cours d'introduction.

1.1.6 Quelques propriétés des flots

1.1.6.1 Unité d'échange avec un flot

Comme cela a été indiqué, l'unité élémentaire d'échange d'information dans un flot est l'octet. Ceci veut dire en particulier que l'opération élémentaire sur un flot d'entrée consiste à « lire » un octet, et que l'opération élémentaire sur un flot de sortie consiste à « écrire » un octet. Nous verrons que nous disposons d'opérations toutefois plus complexes, permettant d'échanger plus d'un octet.

1.1.6.2 Accès séquentiel

Comme son nom le suggère, un flot représente un mécanisme de *producteur-consommateur* : un octet lu dans un flot est *consommé* par le programme et disparaît donc du flot, ce qui signifie que si l'on demande à lire à nouveau un octet, ce sera l'octet suivant du flot qui sera fourni. Cela signifie aussi que le programme doit mémoriser (dans une variable par exemple) tout octet lu s'il veut l'utiliser ultérieurement.

De la même façon, à l'écriture d'un octet dans un flot, cet octet est écrit « à la fin » du flot, et s'ajoute donc aux octets précédemment écrits.

En entrée comme en sortie, lorsque le flot est construit à partir d'un fichier, le système gère une position courante dans le fichier, qui indique l'endroit du fichier où sera lu ou écrit le prochain octet. Attention : lorsqu'un flot est utilisé simultanément en lecture **et** en écriture, le système ne mémorise qu'une seule position ! C'est donc au programmeur de savoir parfaitement où il en est, afin de ne pas lire ou écrire des données à des endroits non convenables du fichier.

Bien sûr, lorsque qu'un flot est construit en lecture seulement à partir d'un fichier externe, le fichier lui-même n'est pas modifié.

On verra que les procédures de lecture et d'écriture dans les flots sont capables de générer des *codes d'erreur* :

- sur un flot en lecture, cette erreur correspond en général à la fin du flot ;
- sur un flot en écriture, cette erreur se produit en général lorsque le flot est « plein » (par exemple, lorsqu'il n'est plus possible d'accroître la taille du fichier sous-jacent par manque de place dans le système de fichiers).

1.1.7 Opérations d'écriture dans un flot

La bibliothèque standard du langage `C` propose de nombreuses opérations d'écriture de caractères ou de chaînes de caractères, faisant référence implicitement ou explicitement à des flots. La table 1.1 décrit quelques opérations d'écriture du langage.

Quelques remarques générales sur ces procédures :

- ces procédures sont en général utilisées en mode « texte » : les octets sont les codes ASCII des caractères dit *imprimables* qui sont transférés vers le flot ;
- les procédures ayant un paramètre de type flot (celles qui commencent par `f` plus `putc`) écrivent sur ce flot ; les autres écrivent sur le flot `stdout` ;

Nom	Prototype	Description
fprintf	int fprintf(FILE *f, const char *d, ...)	Écriture formatée sur le flot f
fputc	int fputc(int c, FILE *f)	Écriture de l'octet c sur le flot f
fputs	int fputs(const char *s, FILE *f)	Écriture de la chaîne s sur le flot f
printf	int printf(const char *d, ...)	Écriture formatée sur stdout
putc	int putc(int c, FILE *f)	Écriture de l'octet c sur le flot f
putchar	int putchar(int c)	Écriture l'octet c sur stdout
puts	int puts(const char *s)	Écriture de la chaîne s sur stdout
sprintf	int sprintf(char *s, const char *d, ...)	Écriture formatée sur la chaîne s

TABLE 1.1 – Quelques opérations de sortie de C

- ces procédures retournent un résultat de type entier ; un résultat égal à -1 (macro EOF définie dans stdio.h) indique une erreur : flot non ouvert en écriture, plus de place pour ajouter les données dans le flot ;

1.1.7.1 Écriture d'octets

Les procédures putc, putchar et fputc permettent d'écrire un octet (c) dans un flot f (ou stdout par défaut pour putchar).

Remarquer que l'octet est transféré sous forme d'un entier : ces procédures effectuent une opération de modulo pour obtenir un entier dans l'intervalle de 0 à 255.

Le résultat en cas de succès de ces procédures est l'octet transmis. On peut tester ce résultat pour détecter une erreur :

```
if (putc('X',f)==EOF) {
  perror("putc a echoue");
  ...
```

Notes :

- les opérations putc et fputc ont un effet strictement identique ; putc est en fait une *macro*, il faut donc prendre garde à des écritures comme putc(i++,f) !
- putchar est également une macro, un appel à putchar(c) correspondant à putc(c,stdout).

1.1.7.2 Écriture de chaînes

L'opération `puts` permet d'écrire une chaîne de caractères complète sur la sortie standard `stdout`. La procédure `fputs` permet d'écrire une chaîne sur un flot passé en paramètre.

Notes :
- `puts` et `fputs` retournent comme résultat le nombre d'octets écrits (et `EOF` en cas d'erreur) ;
- l'opération `puts` *ajoute* un passage à la ligne après impression de son paramètre ; ce n'est pas le cas pour `fputs`.

1.1.7.3 Écriture formatée

Nous avons déjà utilisé à plusieurs reprises l'opération `printf` qui permet d'effectuer des sorties sur la *sortie standard*, qui est habituellement la fenêtre servant de terminal à l'utilisateur. Cette procédure a la syntaxe suivante :

`printf` (*format*, *exp1*, *exp2*, *exp3*...)

La spécification de format est une chaîne de caractères. Le contenu de cette chaîne est recopiée sur la sortie standard, caractère par caractère, sauf lorsque le caractère est un `%`. Dans ce cas, le ou les caractères qui suivent sont des descriptions d'édition :
- `%d` pour afficher un nombre entier sous forme décimale ;
- `%c` pour afficher un nombre entier sous forme caractère ;
- `%x` pour afficher un nombre entier sous forme hexadécimale ;
- `%s` pour afficher une chaîne de caractères ;
- `%f` pour afficher un nombre flottant en virgule fixe ;
- `%e` pour afficher un nombre flottant en virgule flottante, avec un exposant ;
- `%g` pour afficher un nombre flottant en choisissant la meilleure représentation possible (une des deux précédentes) ;
- `%%` enfin, permet l'écriture du caractère '`%`'.

Les directives `%d`, `%c` et `%x` attendent des données de type entier (valeurs déclarées en `char`, `int` et `long`), qui sont converties au format `long` lorsqu'elles sont transmises comme paramètre. Les directives `%f` et `%e` attendent des données de type flottant (valeurs déclarées en `float` et `double`), qui sont converties au format `double` lorsqu'elles sont transmises comme paramètre.

Plusieurs descriptions d'édition peuvent figurer dans le même format, qui vont correspondre aux paramètres *expi*. C'est une erreur que d'avoir un nombre différent de descriptions de format et de paramètres.

Quelques exemples d'utilisation :
- le même nombre, représentant le code ASCII du caractère A, est imprimé sous forme caractère, décimale et hexadécimale ;

  ```
  printf("%c %d %x\n",65,65,65);
  ```

  ce qui donne, à l'exécution :

  ```
  A 65 41
  ```
- deux impressions, selon des formats différents, du même nombre flottant ;

```
printf("%f %e\n",543.21,543.21);
```
ce qui donne, à l'exécution :
```
543.210000 5.432100e+02
```
Entre le caractère `%` et le type d'édition (`cdefsx`, etc), on peut trouver :
- des indicateurs, dont voici certains :

  0 indique que le champ d'édition doit être complété par des 0 plutôt que par des blancs ;

  - indique que la valeur éditée doit s'appuyer à gauche, et non à droite ;

  (un blanc) indique de laisser un blanc devant un nombre positif ;

  + indique qu'un nombre positif doit être édité précédé d'un signe +.

- une taille d'édition minimale ;
- une précision (un nombre, précédé d'un point décimal).
- un modifieur de longueur (caractère `h`, `l`, etc.), permettant de spécifier la longueur (court, long) du nombre à convertir.

Exemples de formats et des sorties correspondantes :
- l'appel
  ```
  printf("*%4d*%4d*\n",33,-33);
  ```
  donne à l'exécution :
  ```
  *  33* -33*
  ```
- l'appel
  ```
  printf("*%4d*%04d*\n",33,33);
  ```
  donne à l'exécution :
  ```
  *  33*0033*
  ```
- l'appel
  ```
  printf("*%-4d*%-4d*\n",33,-33);
  ```
  donne à l'exécution :
  ```
  *33  *-33 *~
  ```
- l'appel
  ```
  printf("*%+4d*%+4d*\n",33,-33);
  ```
  donne à l'exécution :
  ```
  * +33* -33*
  ```
- l'appel
  ```
  printf("*%8.2f*%8.4f*\n",1.531,1.531);
  ```
  donne à l'exécution :
  ```
  *    1.53*  1.5310*
  ```

Notes :
- la procédure `printf` écrit sur la sortie standard `stdout` ; la procédure `fprintf` a un comportement identique, mais écrit sur le flot passé en premier argument ; l'appel à la procédure `printf(format,...)` est donc équivalent à l'appel `fprintf(stdout,format,...)` ;
- les opérations `fprintf`, `printf` et `sprintf`, fournissent un résultat qui est le nombre d'octets effectivement générés dans le flot de sortie, ou -1 (`EOF`) en cas d'erreur ;

Nom	Prototype	Description
fgetc	int fgetc(FILE *f)	Lecture d'un octet sur le flot f
fgets	char *fgets(char *s, int l,FILE *f)	Lecture d'une chaîne sur le flot f
fscanf	int fscanf(FILE *f, const char *d, ...)	Lecture formatée sur le flot f
getc	int getc(FILE *f)	Lecture d'un octet sur le flot f
getchar	int getchar(void)	Lecture d'un octet sur stdin
gets	char *gets(char *s)	Lecture d'une chaîne sur stdin
scanf	int scanf(const char *d, ...)	Lecture formatée sur stdin
sscanf	int sscanf(const char *s, const char *d, ...)	Lecture formatée depuis une chaîne
ungetc	int ungetc(int c, FILE *f)	Rejet d'un octet lu

TABLE 1.2 – Procédures d'entrée sur flots

- le prototype des procédures printf, fprintf et sprintf se terminent par ... : ceci indique que le nombre d'arguments de ces procédures est variable (il dépend du nombre de paramètres à imprimer), et que le compilateur ne doit pas s'offusquer de cela ; le compilateur vérifie néanmoins les premiers arguments (en nombre et en type) ;
- l'opération sprintf, qui n'est pas à proprement parler une opération d'entrée-sortie, puisque l'écriture s'effectue dans un tableau de caractères dont l'adresse est fournie par le programme (le premier paramètre), est décrite ici du fait de ses similitudes avec printf et fprintf.

1.1.8 Opérations de lecture dans un flot

Tout comme le contenu d'un flot de sortie peut être généré octet par octet (par une procédure comme fputc), ou au moyen d'une procédure plus complexe (telle printf), un flot en entrée peut être consommé octet par octet, ou encore au moyen de procédures plus élaborées.

La table 1.2 décrit certaines des opérations s'appliquant à des flots en entrée.

Quelques remarques générales sur ces procédures :

- ces procédures de lecture sont en général utilisées avec des fichiers en mode texte, contenant des caractères imprimables (attention aux caractères qui ne font pas partie du code ASCII : ceux-ci peuvent utiliser des codages différents) ;
- les procédures comportant un paramètre de type flot (celles dont le nom commence par f, plus getc et ungetc) effectuent une lecture sur le flot indiqué ; les autres effectuent une lecture (implicite) sur l'entrée standard stdin ;

- les procédures rendant un résultat de type `int` indiquent une erreur par un résultat égal à -1 (`EOF`) ; les procédures `gets` et `fgets`, rendant un résultat de type `char*` (une chaîne de caractères), indiquent une erreur par un résultat égal à 0 (`NULL`) ;

1.1.8.1 Lecture en mode octet

Les procédures `getc`, `getchar` et `fgetc` permettent de lire un octet dans un flot `f` (ou `stdin` par défaut pour `getchar`).

L'octet lu est rendu comme résultat de ces procédures en cas de succès (sous la forme d'un entier compris entre 0 et 255). On rappelle que la valeur -1 (`EOF`) est retournée en cas d'erreur (plus de données dans le flot, flot non ouvert en lecture).

Notes :

- les opérations `fgetc` et `getc` correspondent à deux réalisations différentes d'une même procédure ; `getc` est en fait une macro ;
- `getchar` est également une macro, un appel à `getchar()` étant fonctionnellement équivalent à `getc(stdin)` ;
- on insiste sur le fait que ces trois procédures retournent une valeur de type `int` ! C'est donc une **erreur** que de stocker le résultat dans une variable de type `char` : le caractère de code ASCII 255 est alors assimilé à l'erreur -1 (`EOF`) !
- l'opération `ungetc` permet de rejeter un octet, qui est « réintroduit » dans le flot d'entrée, et sera fourni à nouveau lors d'une nouvelle lecture d'octet. Cette opération a pour but de permettre à un programme d'analyser un flot qui est lu octet par octet par `getc`, puis, par exemple lorsque le premier octet d'un nombre est reconnu, de rejeter cet octet pour effectuer une nouvelle lecture, cette fois au moyen d'une procédure spécialisée comme `scanf`. On notera que la norme du langage précise que l'octet rejeté *doit* être le dernier octet lu, et qu'*un seul octet* peut être ainsi rejeté ;
- noter qu'il n'existe pas de macro `ungetchar`...

1.1.8.2 Lectures de chaînes

Les procédures `gets` et `fgets` permettent de lire une chaîne de caractères dans un flot (le flot passé en argument pour `fgets`, `stdin` pour `gets`).

La lecture dans le flot s'effectue jusqu'à ce que :

- le caractère « fin de ligne » soit lu ;
- il n'y ait plus aucun octet dans le flot (fin de fichier) ;
- dans le cas de `fgets`, on ait lu `n-1` octets

Dans tous les cas, la chaîne de caractères est « terminée » par le caractère ASCII de code 0.

Notes :

- ces deux procédures lisent des « lignes », qui se terminent par un passage à la ligne (format dit « texte » sous Windows) ;

- l'opération `gets(tab)` est considérée comme potentiellement dangereuse [2], car elle lit une chaîne de caractères (qui peut être de très grande taille) et la recopie dans un tableau, `tab` avec un risque de débordement du tableau. On lui préférera une expression telle que `fgets(tab,sz,stdin)` qui permet de spécifier la taille `sz` du tableau passé en paramètre.

1.1.8.3 Lecture formatée

La procédure `scanf` permet de lire le flot d'entrée `stdin`, en décodant les octets de ce flot selon des spécifications décrites par un format. Ces spécifications sont très voisines de celles qui sont utilisées par `printf`. La syntaxe de l'opération est :

`scanf(`*format*`, ...)`

Un format est une chaîne de caractères, qui va être « comparée » avec les octets lus sur le flot d'entrée de la façon suivante :

- un « blanc » dans le format accepte un nombre arbitraire, éventuellement nul, de « blancs » en entrée. Les blancs sont l'espace proprement dit, de code ASCII 32, le passage à la ligne, de code 10, le caractère de tabulation, de code 8, etc.
- un caractère « standard » accepte un caractère identique du flot d'entrée.
- un caractère `%` introduit une description de lecture. Voici quelques exemples :
  - ▷ `%d` attend un entier au format décimal éventuellement précédé de blancs, puis d'un signe.
  - ▷ `%x` attend un entier au format hexadécimal (« chiffres » de `0` à `9` et `A` à `F`).
  - ▷ `%c` attend un caractère unique ; le résultat est le code ASCII du caractère.
  - ▷ `%e` ou `%f` attendent un flottant, éventuellement précédé de blancs. Le nombre peut être représenté avec signe, partie entière, décimale et exposant.
  - ▷ `%s` attend une séquence de caractères non blancs ; la lecture s'arrête au premier blanc rencontré.
  - ▷ `%[`*liste-de-caractères*`]` attend une séquence composée des caractères de la liste.
  - ▷ `%%` attend un caractère `%`.

Lorsqu'une spécification `%d`, `%x`, `%c`, `%e`, `%f`, etc. a décodé un nombre, le résultat est placé dans la cellule mémoire dont l'adresse a été fournie comme paramètre complémentaire de `scanf` : cette cellule mémoire peut être l'adresse d'une variable, d'un élément de tableau, d'un champ de structure... Il faut fournir une telle adresse pour chaque description de lecture du format.

Si la variable est de type `long`, et non `int`, la spécification de conversion doit s'écrire `%ld` ou `%lx`.

Si la variable est de type `short`, et non `int`, la spécification de conversion doit s'écrire `%hd` ou `%hx`.

Si la variable est de type `double`, et non `float`, la spécification de conversion doit s'écrire `%le` ou `%lf`.

Le résultat de la procédure est un entier indiquant le nombre effectif de variables qui ont pu être lues par le programme, ou `EOF` en cas d'erreur de lecture, de fin de fichier rencontrée, etc.

---

2. elle a donné lieu à beaucoup de failles de sécurité

Exemple Le programme suivant montre la procédure en action :

```
#include <stdio.h>

int main (int argc,char *argv[])
{
int n,v,w,x;

  v=w=x=-1;
  n=scanf("a %d b %d c %d d",&v,&w,&x);
  printf("n=%d,v=%d,w=%d,x=%d\n",n,v,w,x);
  return 0;
}
```

Voici quelques exemples d'exécution, l'entrée fournie au programme précédant la réponse de celui-ci :

```
$ prog
a 333 x 222 u 444
n=1,v=333,w=-1,x=-1
$ prog
a55b66 c 222d
n=3,v=55,w=66,x=222
$ prog
u23
n=0,v=-1,w=-1,x=-1
$ prog
a 0x2F4 b 333 c 444 w
n=1,v=0,w=-1,x=-1
```

Notes :

- la procédure `fscanf` permet d'opérer des lectures depuis un flot dont la référence est donnée comme premier paramètre (c.f. l'exemple du paragraphe 1.1.11, page 22) ; elle a un comportement identique à celui de `scanf` ; un appel à `scanf(...)` équivaut à `fscanf(stdin,...)` ;
- la procédure `sscanf` permet la lecture formatée à partir d'une chaîne de caractères, et donc l'interprétation d'une suite de caractères en tant que nombre :
  `sscanf("743.25","%f",&v);`
  affecte la valeur flottante `743.25` à la variable `v` (déclarée en `float`) ;
- les trois procédures `scanf`, `fscanf` et `sscanf` rendent un résultat qui est le nombre de paramètres correctement lus (attention : ce n'est pas le nombre de caractères lus !), ou -1 (`EOF`) en cas d'erreur.
- on notera que les prototypes de ces trois procédures se terminent par `...`, ce qui indique que le nombre d'arguments est variable : il dépend en effet du nombre de paramètres à lire.

1.1.9 Un exemple : « head »

On se propose d'écrire le programme « `head` », qui lit les premières lignes (au plus 10) de l'entrée standard, et les écrit sur la sortie standard [3]. Voici le programme source :

```
#include <stdio.h>

#define SIZE 4000

int main (int argc,char *argv[])
{
int i;
char buffer[SIZE];
char *res;

for (i=0;i<10; i++) {
  res=fgets(buffer,SIZE,stdin);
  if (res==NULL)
    break;
  else
    fputs(buffer,stdout);
  }

}
```

Notes

- nous utilisons ici une facilité du préprocesseur pour définir une *macro* :
  `#define SIZE 4000`
  Cette ligne associe à la variable du préprocesseur `SIZE` la suite des 4 caractères « `4000` ». Chaque fois que le préprocesseur rencontre cette variable dans le texte source sur programme, ce qu'il fait à deux reprises, il la remplace par sa valeur, les 4 caractères « `4000` ». Tous se passe comme si l'on avait écrit :
  `char buffer[4000];`
  et
  `res=fgets(buffer,4000,stdin);`
  aux deux endroits où cette variable est utilisée. Cette technique simplifie la maintenance d'un programme, en permettant, comme ici, de rendre cohérentes déclarations et utilisations.
- de même, la variable du préprocesseur `NULL`, est utilisée ici pour tester la valeur rendue par la procédure `fgets()` (cette variable est définie dans le fichier `stdio.h`).
- on a utilisé la procédure `fgets` permettant de lire (de façon plus sûre que `gets`) des lignes dans le flot d'entrée ; on pourra étudier, à titre d'exercice, ce qui se passe si l'une des lignes d'entrée contient plus de `SIZE` caractères.

---

3. voir le manuel `man head`

1.1.10 Autres procédures

Nous allons présenter ici quelques procédures supplémentaires liées aux mécanismes d'entrées-sorties.

1.1.10.1 Gestion des erreurs

Les entrées-sorties sont une source fréquente d'erreurs. Afin de pouvoir préciser quelle type d'erreur est intervenue, la bibliothèque standard du `C` offre les mécanismes suivants :

- une variable *globale* nommée `errno`, qui contient un numéro représentant le type de la dernière erreur intervenue ;
- une procédure `perror(3)`, qui est destinée à imprimer sur le flot de gestion des erreurs un message représentatif (en clair) de la dernière erreur intervenue.

À noter que l'utilisation de la variable externe `errno` nécessite l'inclusion du fichier d'en-tête `errno.h`. À noter également que la variable `errno` ne concerne pas uniquement les entrées-sorties, mais toutes les procédures de bibliothèque du `C`, ainsi que les appels aux procédures du système.

On trouvera ci-dessous quelques macro-définitions de numéros d'erreurs, avec le message (anglais) associé ainsi qu'une traduction approximative :

Erreur	Message	Signification
E2BIG	Arg list too long	la liste des arguments du programme est trop longue
EBADF	Bad file descriptor	descripteur de flot incorrect
EEXIST	File already exists	le fichier existe déjà
EIO	I/O error	erreur d'entrée-sortie
ENOENT	No such file or directory	le fichier ou le répertoire indiqué n'existe pas
EPERM	Permission denied	le programme n'a pas les droits requis pour l'opération demandée
EROFS	Read-only filesystem	le système de fichiers est en lecture seule

La liste complète des codes d'erreur est accessible dans la page de manuel nommé `intro` de la section 2, et est donc consultable par la commande :

```
man -s 2 intro
```

Le prototype de la procédure `perror` est le suivant :

```
void perror (char *msg);
```

La procédure `perror` se comporte comme suit :

- si l'adresse passée en argument n'est pas nulle, et ne désigne pas une chaîne de caractères ne contenant que le caractère `0` (chaîne vide), la chaîne de caractères désignée est imprimée telle quelle (sans formatage) suivie d'un caractère `:` sur le flot de gestion des erreurs (`stderr`) ;
- un message descriptif de la dernière erreur rencontrée est ensuite imprimé, suivi d'un saut de ligne, toujours sur `stderr`.

Note importante : la variable `errno` n'est pas modifiée par un appel qui réussit, c'est donc bien toujours la dernière erreur rencontrée par le programme qui est imprimée.

1.1.10.2 Autres procédures de manipulation de flots

On dispose des procédures suivantes :

- `int feof (FILE *pf);` retourne une valeur non nulle (vraie) lorsqu'une fin de fichier a été détectée sur le flot `pf`. Attention : cet indicateur devient non nul **après** qu'un essai de lecture infructueux a été fait !
- `int ferror (FILE *pf);` retourne une valeur non nulle (vraie) lorsqu'une erreur a été détectée sur le flot `pf`. Attention : comme pour le précédent, cet indicateur devient non nul **après** qu'une erreur a été effectivement détectée !
- `void clearerr (FILE *pf);` remet à zéro (faux) les deux indicateurs de fin de fichier et d'erreur sur le flot `pf`.

1.1.10.3 Autres procédures de manipulations de fichiers

On dispose dans la bibliothèque standard du `C` de procédures permettant d'accéder au système de gestion des fichiers, de façon indépendante du système sous-jacent.

Voici quelques-unes de ces procédures :

- `int remove (char *nom);` détruit le fichier dont le nom est passé en argument, rend 0 si le fichier a été détruit, -1 sinon (erreur dans `errno`) ;
- `int rename (char *old,char *new);` renomme le nom du fichier `old` sous le nom `new`, rend 0 si l'appel a réussi, -1 sinon (erreur dans `errno`) ;
- `FILE *tmpfile (void);` crée un flot vers un fichier temporaire en mode lecture et écriture, le fichier est détruit lorsque le flot est fermé ; l'appel rend `NULL` s'il échoue (erreur dans `errno`).

1.1.11 Un exemple : « high scores »

On veut, dans un programme, récupérer dans un fichier de nom « `scores` » deux valeurs, un entier et un nom. Le format du fichier « `scores` » est le suivant : un entier sous forme décimale, un caractère `*` servant de séparateur, et une suite de caractères constituant le nom. Le nom est composé de lettres, chiffres, et espaces. Voici un exemple de contenu du fichier :

```
623147*The terminator
```

Voici un exemple de programme réalisant les opérations désirées :

```
#include <stdio.h>

int main (int argc,char *argv[])
{
int high;
char ident[128];
```

```
FILE *hs;
int cr;

  hs=fopen("scores","r");
  if (hs==NULL) {
    perror("Unable to open the \"scores\" file");
    return 1;
  }
  cr=fscanf(hs,"%d*%[0-9a-zA-Z ]",&high,ident);
  if (cr!=2) {
    fprintf(stderr,"Incorrect \"scores\" file format\n");
    return 1;
  }
  printf("Best score : %s : %d\n",ident,high);
  fclose(hs);
  return 0;
}
```

On note :

- le format utilisé dans `fscanf` – `[0-9a-zA-Z ]` – s'interprète comme : « suite éventuellement nulle de caractères appartenant à l'intervalle (ASCII) de `'0'` à `'9'` (chiffres décimaux), ou de `'a'` à `'z'` (lettres minuscules) ou de `'A'` à `'Z'` (lettres majuscules), ou encore espace »
- si `fscanf` ne rend pas la valeur attendue (2 car 2 variables sont lues en cas de fonctionnement normal), on n'utilise pas la procédure `perror` : c'est une erreur de format du fichier, pas une erreur du système d'entrées-sorties.

1.1.12 Un exemple, « head, 2 »

On se propose de modifier le programme `head` pour qu'il travaille sur un fichier dont le nom lui est transmis comme paramètre, au lieu de `stdin`.

```
#include <stdio.h>

#define SIZE 4000

int main (int argc,char *argv[])
{
char buffer[SIZE];
/* la macro FILENAME_MAX est definie dans stdio.h */
/* elle donne le nombre max de caracteres d'un */
/* "pathname", c-a-d le nom absolu d'un fichier */
char name[FILENAME_MAX];
FILE *hs;
```

```
int i;char *res;

  /* si on n'a pas donne d'argument, on stoppe ! */
  if (argc != 2) {
    printf("File name missing\n");
    return 2;
  }
  /* on recopie le nom du fichier, en ne prenant pas */
  /* plus que les FILENAME_MAX-1 premiers caracteres */
  strncpy(name,argv[1],FILENAME_MAX-1);
  /* on n'oublie pas le '\0' final ! */
  name[FILENAME_MAX-1]=0;

  /* on ouvre le fichier "name" en lecture seule ; en */
  /* cas d'erreur, on affiche un message et on stoppe */
  hs=fopen(name,"r");
  if (hs==NULL) {
    perror(name);
    return 1;
  }

  for (i=0; i<10; i++) {
    res=fgets(buffer,SIZE,hs);
    if (res==NULL)
      break;
    else
      fputs(buffer,stdout);
  }
  fclose(hs);
  return 0;
}
```

1.2 Compléments de cours

Le lecteur avisé trouvera ici quelques compléments sur le cours, répondant à des questions pouvant surgir sur le fonctionnement précis des mécanismes d'entrées-sorties en `C`.

1.2.1 Caractères et chaînes de caractères en `C`

1.2.1.1 Le type `char`

Dans le langage `C`, on dispose du type prédéfini `char`, que l'on traduit souvent par *caractère*. Il faut bien comprendre que ce type permet simplement de représenter un entier sur 8 bits, soit

256 valeurs. En général, le type `char` est « signé » (codage sur 8 bits en complément à 2) et permet donc de représenter les valeurs entières comprises entre -128 et +127.

Il existe une version non-signée : c'est le type `unsigned char`. Ce type permet de représenter les entiers sur 8 bits, donc les valeurs comprises entre 0 et +255.

Dans certains environnements toutefois (et la norme ANSI du langage `C` l'autorise), le type `char` est non-signé : les valeurs représentables sont ainsi les entiers compris entre 0 et +255. La norme ANSI prévoit même un troisième cas de figure : le traitement du type `char` en tant que « pseudo non signé » (voir la norme ANSI pour plus de précisions).

1.2.1.2 Les constantes caractères

Le langage `C` accepte l'utilisation de constantes (entières) définies par un caractère : dans ce cas, la valeur numérique est définie comme étant le code ASCII de ce caractère (sous forme d'un entier de type `int`). Ainsi, écrire dans un programme :

```
int c;
c='A';
```

est totalement équivalent à écrire :

```
int c;
c=65;
```

ou encore :

```
int c;
c='\101';
```

ou encore :

```
int c;
c='\x41';
```

car le caractère A (a majusucule) correspond au code ASCII 65, qui s'écrit en octal $101 = 1 \times 8^2 + 1$ et en hexadécimal $41 = 4 \times 16 + 1$. En général, on utilise la notation octale ou hexadécimale uniquement pour les caractères non autorisés dans le code source du programme (caractères de contrôle).

On rappelle par ailleurs que certains caractères ont un codage spécial, introduit par le caractère *backslash* \ :

`a` : ***a**lert*, bip sonore ;

`b` : ***b**ackspace*, espace arrière ;

`f` : ***f**ormfeed*, saut de page ;

`n` : ***n**ewline*, saut de ligne ;

`r` : *carriage **r**eturn*, retour chariot en début de ligne ;

`t` : *horizontal **t**ab*, tabulation horizontale ;

v : *vertical tab*, tabulation verticale ;

\ : *backslash* ;

' : simple quote ;

" : double quote ;

? : point d'interrogation ;

1.2.2 Chaînes de caractères

1.2.2.1 Le type chaîne de caractères

Pour représenter et stocker une séquence de caractères (un mot, une phrase), le langage C ne propose pas de type de données particulier : il utilise simplement la notion de tableau (plus précisément, un tableau de `char`) pour stocker ces séquences. Une telle séquence porte le nom de *chaîne de caractères* ou *string* en anglais.

Comme le nombre de caractères contenus dans le tableau peut être variable, le langage C utilise la convention suivante : toute chaîne est terminée par un caractère dit *nul*, c'est-à-dire de valeur 0 (le code ASCII correspondant est `NUL`). Ainsi, les nombreuses procédures de bibliothèque qui utilisent ce type de données admettent des paramètres de type `char*` c'est-à-dire « adresse de caractère », soit en réalité « adresse de premier caractère d'un tableau ».

1.2.2.2 Constantes chaînes de caractères

Le langage C utilise une notation particulière pour les chaînes de caractères : on écrit la séquence de caractères entre deux caractères double-quote. Par exemple :

```
"une chaine simple"
"une chaine avec un \n saut de ligne au milieu"
"\001\002\003\x04\x05"
```

Si l'on a une chaîne de caractères assez longue, et que l'on souhaite (pour des questions de lisibilité, par exemple) la fractionner sur plusieurs lignes, il faut l'écrire comme suit :

```
printf("Cette chaine de caracteres est vraiment trop "
       "longue pour etre ecrite sur une seule ligne \n"
       "et on peut donc l'ecrire comme ca !\n");
```

Il est en effet interdit d'intégrer un saut de ligne à l'intérieur d'une chaîne de caractères (un saut de ligne dans le code source, pas le caractère particulier `\n` qui est un caractère autorisé, voir l'exemple ci-dessus).

Le compilateur traite alors cette définition en effectuant une *concaténation* : il alloue un seul espace mémoire pour stocker la totalité des octets (et les caractères nuls intermédiaires sont supprimés).

Ces constantes peuvent être utilisées :

- comme valeur d'initialisation pour un tableau de caractères (dont la taille est précisée ou non) ;

∘ à n'importe quel endroit où l'on pourrait utiliser un variable de type `char*`.
Par exemple :

```
char tab1[10] = "coucou";
char tab2[] = "salut";
char *str="hello";
```

Dans le premier cas, le compilateur alloue une zone mémoire pour le tableau `tab1` de 10 octets, le premier élément étant le caractère `'c'` (code ASCII 99), le second étant le caractère `'o'` (code ASCII 111), le sixième étant le caractère `'u'`, le septième étant le caractère `NUL` (valeur numérique 0), les 3 éléments restants étant indéterminés.

Le second est à peu près identique, sauf que le compilateur détermine lui-même la taille du tableau à allouer : cette taille comprend le caractère NUL final (donc une taille totale de 6 caractères dans cet exemple).

Enfin, dans le troisième cas, le compilateur alloue pour la variable `str` une zone mémoire permettant de stocker une adresse (4 octets sur un système 32 bits, 8 octets sur un système 64 bits). Il alloue ensuite une zone mémoire *anonyme* (aucune variable ne désigne cette zone mémoire) de 6 octets (au moins) qui contient les caractères `'h'`, `'e'`, `'l'`, `'l'`, `'o'` et le caractère NUL final : l'adresse de cette zone mémoire est alors stockée dans la variable `str`.

1.2.2.3 Particularités

Lorsque l'on utilise une variable de type tableau de caractères, il est possible de l'initialiser (lui donner une valeur au moment de sa déclaration) mais, comme pour tous les types tableaux en `C`, on ne peut affecter « globalement » un tableau.

```
char tab[] = "coucou" ; /* initialisation : OK */
...
tab = "hello" ; /* affectation : INCORRECT */
```

Si l'on utilise une variable de type adresse, on peut l'affecter (attention : on modifie bien la variable de type adresse et non pas le contenu du tableau désigné !).

```
char *str = "coucou" ; /* initialisation : OK */
...
str = "hello" ; /* affectation : OK */
```

Attention toutefois : la première zone mémoire est définitement perdue après la modification de la variable `str` !

Attention enfin aux comparaisons sur les chaînes de caractères : les opérateurs arithmétiques ne peuvent comparer que les adresses et pas le contenu des zones mémoires correspondantes !

```
char tab1[] = "coucou" ;
char *str = "coucou" ;
...
if (tab1 == str) /* ce test est toujours faux */
```

```
...
if (tab1 == "coucou") /* ce test est toujours faux */
...
if (str == "coucou") /* ce test est faux en general */
                     /* mais sera parfois vrai !! */
if (strcmp(str,tab1)==0) /* ceci est la bonne facon */
                         /* d'ecrire le test !! */
```

1.2.3 Fichiers texte et fichiers binaires

Le système d'exploitation Linux, comme la quasi-totalité des Unix, ne connaît qu'un seul type de fichier : le fichier dit *régulier*, considéré comme une succession d'octets. On peut donc utiliser, sur un même flot, aussi bien des procédures de type caractère (comme `putc`, `getc`, `printf`) que des procédures de type binaire (comme `fread` ou `fwrite`).

Dans le cas du système Windows, il y a distinction entre deux types de fichiers :
- les fichiers binaires, qui ressemblent aux fichiers réguliers sous Linux ;
- les fichiers texte.

Un fichier texte a une structure particulière :
- il est composé de lignes, chaque ligne étant terminée par une combinaison de **deux caractères**, qui sont les codes ASCII 13 (`Control-M` ou *Carriage Return* ou *Retour chariot*) et 10 (`Control-J` ou *Line Feed* ou *Saut de ligne*) [4] ;
- il est terminé, après la dernière ligne, par un caractère de fin de fichier qui est le code ASCII 26 (`Control-Z` ou *Substitute* ou *Substitution*) [5].

Lors de la construction d'un flot par `fopen` sous Windows, le fichier est supposé par défaut être de type texte. Si l'on veut travailler avec un fichier en mode binaire, on doit le préciser en ajoutant au mode (le deuxième paramètre de `fopen`) un caractère `b`. Voici un exemple de manipulations :

```
FILE *fic_texte,*fic_bin;
/* ouverture d'un fichier texte */
fic_texte=fopen("toto.txt","r");
if (fic_texte==NULL)
{
  perror("toto.txt");
  exit(2);
}
/* ouverture d'un fichier binaire */
fic_bin=fopen("toto.dat","wb");
...
```

4. cette convention provient de la gestion des imprimantes, où le passage à la ligne suivante nécessitait deux opérations : ramener le chariot portant la tête d'impression en début de ligne, puis faire tourner le rouleau entraînant le papier d'une ligne.

5. toujours sur les imprimantes, ce caractère indiquait un changement d'alimentation de papier.

Note : sous Linux, le `b` éventuel en fin de mode est tout simplement ignoré.

Il est recommandé de manipuler les fichiers binaires avec `fread` et `fwrite` (décrites ci-après), et les fichiers texte avec les entrées-sorties caractère, chaîne ou formatées comme `fprintf`, `fscanf`, `fputs`, `fgets`, `fputc`, `fgetc`...

1.2.3.1 Écriture en format binaire

Il est donc possible d'écrire dans un flot des données dite « binaires », car directement issues de la représentation interne en mémoire des données utilisateurs.

Cette opération d'écriture se fait grâce à la procédure `fwrite` dont le prototype est :

```
size_t fwrite(void *addr,size_t sz,size_t nb,FILE *pf);
```

Cette procédure permet d'écrire dans le flot désigné par `pf` les données en mémoire à partir de l'adresse désignée par `addr`, et représentant `nb` objets contigus, chacun étant de taille `sz`.

Note : la syntaxe particulière `void *addr` permet de désigner en `C` une adresse mémoire générique. En effet, en toute rigueur, le type adresse dépend de la nature de l'objet désigné (adresse d'entier, adresse de flottant, adresse de caractère...) ; pour éviter d'avoir une procédure d'écriture pour chaque type d'adresse, on utilise ce type générique qui permet d'unifier tous les types d'adresses. Noter aussi l'utilisation du type défini `size_t`, désignant une taille de donnée : selon les systèmes, c'est un entier sur 32 bits ou 64 bits, mais son nom est aussi standardisé.

Voici quelques exemples d'utilisation de cette procédure :

```
FILE *pf;
int tab[10];
char *str="Ceci est une chaine";

/* on ouvre le fichier "toto.out" */
pf=fopen("toto.out","w");
if (pf==NULL) {
  perror("toto.out");
  exit(2);
}
/* ecriture du tableau de 10 entiers */
if (fwrite(tab,sizeof(int),10,pf)!=10) {
  perror("fwrite");
  fclose(pf);
  exit(2);
}
/* ecriture d'une chaine de caracteres */
if (fwrite(str,1,strlen(str),pf)!=strlen(str)) {
  perror("fwrite");
  fclose(pf);
  exit(2);
```

```
}
/* ecriture de l'adresse memoire de str */
if (fwrite(&str,sizeof(char*),1,pf)!=1) {
  perror("fwrite");
  fclose(pf);
  exit(2);
}
...
```

On note que le résultat de la procédure `fwrite` est le nombre d'objets écrits ; si l'appel a réussi, ce résultat est donc identique au troisième argument (une différence indique une erreur, d'où les tests dans l'exemple ci-dessus).

1.2.3.2 Lecture en format binaire

On dispose également du symétrique de la procédure `fwrite`, appelée `fread`. Son prototype est :

```
size_t fread (void *addr,size_t sz,size_t nb,FILE *pf);
```

Le résultat rendu par la procédure est le nombre d'éléments effectivement lus (ce nombre est toujours positif ou nul et toujours inférieur ou égal à `nb`). Voici quelques exemples d'utilisation de cette procédure :

```
FILE *pf;
int tab[10];
char str[20];
size_t nb,strsz;
double d;
int i;

pf=fopen("toto.dat","r");
if (pf==NULL) {
  perror("toto.dat");
  exit(2);
}
nb=fread(tab,sizeof(int),10,pf);
for (i=0;i<nb;i++)
   (void) printf("tab[%d]=%d\n",i,tab[i]);
strsz=fread(str,sizeof(char),19,pf);
str[strsz]=0;
puts(str);
if (fread(&d,sizeof(double),1,pf)==1)
   (void) printf("d=%f\n",d);
fclose(pf);
...
```

Note : dans le cas d'une lecture de chaîne de caractères avec `fread`, il faut bien prendre garde à :

- lire un caractère de moins que la taille réservée pour la chaîne ;
- ajouter ensuite le `0` final.

La procédure `fread` transfère des octets en mémoire, et ne connaît donc absolument rien de la structuration de ces octets.

1.2.4 Gros-boutistes et petits-boutistes

Les deux termes ci-dessus sont des libres traductions [6] des termes anglais *big-endian.* et *little-endian* . De la même façon que certains écrivent de gauche à droite alors que d'autres écrivent de droite à gauche (ou de bas en haut), on trouve plusieurs façons de coder les entiers sur plus de 1 octet :

- les gros-boutistes écrivent d'abord les octets de poids fort (*d'abord* représente ici un arrangement en mémoire) et ensuite les octets de poids faible ;
- les petits-boutistes écrivent d'abord les octets de poids faible et ensuite les octets de poids fort.

Par exemple, pour coder la valeur décimale 4219 sur 2 octets à partir de l'adresse mémoire N :

- un gros-boutiste la représentera par la séquence `0x107B`, ce qui signifie encore que la cellule mémoire à l'adresse N contient la valeur `0x10` (en décimal, 16=4219/256), l'adresse N+1 contenant la valeur `0x7B` (en décimal, 123=4219 mod 256) ;
- un petit-boutiste la représentera par la séquence `0x7B10`, ce qui signifie encore que la cellule mémoire à l'adresse N contient la valeur `0x7B`, l'adresse N+1 contenant la valeur `0x10`.

Rappelons que le processeur Intel Pentium (et toutes ses variations) est un processeur petit-boutiste, et que par conséquent, les systèmes d'exploitation Windows et Linux sur PC à base de processeur Intel sont petits-boutistes. La majorité des autres processeurs, comme le Motorola PowerPC sur MacIntosh, le Sun Sparc sur stations Sun, les MIPS R10000, R12000 sur stations SGI, le HP-PA sur stations Hewlett-Packard, et donc les systèmes associés, sont gros-boutistes.

Avertissement :

> Il a été indiqué que les procédures d'écriture et de lecture binaires `fwrite` et `fread` transféraient directement les représentations internes en mémoire : on obtient donc des résultats différents selon que l'on est sur un système gros-boutiste ou un système petit-boutiste !
>
> En cas de transfert d'un fichier binaire sur un autre environnement, on n'est pas du tout sûr que le même programme `C`, compilé sous le nouvel environnement, saura relire les données du programme compilé sous l'environnement initial. Ceci peut donc générer de gros problèmes de portabilité des fichiers binaires entre systèmes de conventions différentes.

6. inspirées des *Voyages de Gulliver* de J. Swift ; certains préfèrent les termes gros-boutien et petit-boutien

C'est pour cette raison que, si l'on cherche la portabilité des programmes développés, un grand soin doit être apporté à la manipulation des fichiers de données binaires (détection du type de convention adoptée, procédures adaptées).

1.2.5 Tampons en mémoire

On a dit que les flots correspondaient en général à un fichier du système de fichier. Il serait cependant très inefficace pour le fonctionnement du programme de devoir physiquement lire ou écrire sur le disque dur à chaque fois que l'on lit ou que l'on écrit un octet dans un flot : une entrée-sortie physique peut prendre quelques milli-secondes, ce qui est très long par rapport à une instruction machine de quelques nano-secondes !

Pour cela, la bibliothèque standard du `C` associe à chaque flot un tampon mémoire (le terme anglais de *buffer* est largement plus utilisé que le terme français...). La plupart des entrées-sorties consistent alors simplement à écrire ou lire des caractères dans le buffer, l'accès physique (en lecture ou en écriture) étant réalisé lorsque :

- on réalise une opération d'écriture sur le flot amenant le tampon à être plein ;
- on réalise une opération de lecture sur le flot amenant le tampon à être vide ;
- le programme le demande explicitement par un appel à la procédure
  ```
  int fflush (FILE *pf);
  ```
- d'autres circonstances détaillées ci-après.

De plus, chaque flot peut avoir trois modes de fonctionnement :

- un mode dit « bloc » (ou *block-buffered*) ; les tampons sont de taille `BUFSIZ` (comme d'habitude, défini dans `stdio.h`, valant usuellement 4096), un tampon est plein lorsque les `BUFSIZ` caractères ont été écrits dans le tampon, il est vide lorsque les `BUFSIZ` caractères ont été lus ;
- un mode dit « ligne » (ou *line-buffered*) ; les tampons sont toujours de taille `BUFSIZ`, mais sont également écrits dès qu'une fin de ligne est écrite dans le tampon (flots en écriture), et les lectures se font par lignes, c'est-à-dire par séquence d'au plus `BUFSIZ` caractères suivis d'un caractère de fin de ligne ;
- un mode dit « brut » ; dans ce cas, le tampon n'est pas utilisé et toute lecture ou écriture donne lieu à une écriture ou une lecture physique.

Par défaut, les flots prédéfinis `stdin` et `stdout` sont en mode ligne, le flot `stderr` est en mode brut, les flots construits à partir de fichiers sont en mode bloc.

1.2.6 Positionnement dans un flot

Certains flots, et notamment les flots construits à partir de fichiers, sont dits « positionnables » (traduction approximative du terme anglais *seekable*), c'est-à-dire que l'on peut associer au flot une notion de position courante. Cette position correspond dans ce cas à un décalage en octets (ou caractères) par rapport au début du fichier. On rappelle que toute lecture ou écriture se fait alors à la position courante dans le flot (et que la position courante est modifiée par l'opération de lecture ou d'écriture).

On peut obtenir la position dans un flot par la procédure :

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

position que l'on peut ultérieurement utiliser dans une autre procédure :

```
int fseek (FILE *pf,long pos,int whence);
```

Dans cette dernière procédure, le dernier paramètre permet d'indiquer que la position est :
- relative au début du fichier si le dernier paramètre est `SEEK_SET` ;
- relative à la position courante si le dernier paramètre est `SEEK_CUR` ;
- relative à la fin du fichier si le dernier paramètre est `SEEK_END`.

les trois valeurs `SEEK_SET`, `SEEK_CUR` et `SEEK_END` étant définies comme d'habitude dans `stdio.h`.

On dispose également de la procédure :

```
void rewind (FILE *pf);
```

dont l'appel produit une action identique à :

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

c'est-à-dire un repositionnement au début du fichier.

1.3 Travaux pratiques

1.3.1 Arguments de la procédure `main`

Écrire un programme en `C` qui affiche chacun de ses arguments comme sur l'exemple ci-dessous :

```
$ ./affarg truc bidule 123
./affarg a 4 arguments :
arg[0] = "./affarg"
arg[1] = "truc"
arg[2] = "bidule"
arg[3] = "123"
```

1.3.2 Consulter le manuel

Lire attentivement la page de manuel de la procédure `strcmp(3)` puis répondre aux questions suivantes :
- quel fichier d'en-tête doit-on inclure dans un programme pour utiliser sans ennui cette procédure ?
- préciser le nombre et le type des paramètres de cette procédure ;
- quel type de résultat rend cette procédure ?
- quel est le fonctionnement précis de cette procédure ?

Ecrivez un programme nommé `strcmp`, admettant deux arguments, et qui rend comme *status code* 0 si les deux arguments sont identiques, 1 si le premier argument est inférieur (au sens de l'ordre lexicographique) au second, 2 sinon.
NB1 : le programme `strcmp` n'affiche rien !
NB2 : dans une fenêtre de terminal, on peut consulter le status code d'un programme en utilisant la commande `echo` et la pseudo-variable `$?` :

```
$ strcmp toto toto
$ echo $?
0
$ strcmp bidule toto
$ echo $?
1
$ strcmp toto bidule
$ echo $?
2
```

1.3.3 Conversion de nombres

Lisez le manuel de la procédure `atoi(3)` permettant de convertir une chaîne de caractères représentant un nombre entier écrit en décimal, en une valeur entière égale à ce nombre. Puis écrivez un programme qui affiche à l'écran le nombre entier 2147483647 ainsi que le nombre entier obtenu par la conversion de la chaîne de caractères `"2147483647"` par la procédure `atoi(3)`.
Puis remplacez 2147483647 par 2147483648 et commentez le résultat obtenu.

NB : ces valeurs sont codées *en dur* dans le programme.

1.3.4 Écriture d'entiers

Écrire un programme en `C`, admettant un argument qui est un entier `N`, et qui affiche tous les entiers de `1` à `N`, chacun sur une ligne distincte, sur la sortie standard. Voici un exemple de fonctionnement :

```
$ ./affint 4
1
2
3
4
```

NB : bien traiter le cas où le nombre de paramètres est incorrect, le cas où le paramètre n'est pas un entier, le cas où c'est un entier négatif ou nul. Le *status code* du programme doit être 0 en cas de réussite, 1 si le paramètre est manquant ou incorrect.

Puis modifier le programme précédent de façon que, s'il est lancé sans argument, il prenne comme valeur de `N` la valeur par défaut 10.

Enfin, modifier le programme de façon à ce que l'on puisse donner en dernier paramètre (après l'éventuelle valeur de N) le nom d'un fichier dans lequel l'écriture des entiers doit être faite.

```
$ ./affint 4
1
2
3
4
$ ./affint 4 toto.txt
$ cat toto.txt
1
2
3
4
$ ./affint 4 /toto.txt
/toto.txt: Permission denied
```

NB : prendre soin à bien traiter les erreurs d'entrées-sorties, toute erreur devant conduire à un *status code* égal à 2.

1.3.5 Nombre de caractères et de lignes dans un fichier

Écrire un programme en C qui compte le nombre de caractères et le nombre de lignes d'un fichier, le nom du fichier étant l'unique paramètre de la ligne de commande :

```
$ ./compte toto.txt
toto.txt: 15 caracteres, 4 lignes
$ ./compte /toto.txt
/toto.txt: No such file or directory
```

Modifier le programme précédent afin que l'on puisse effectuer l'opération sur plusieurs fichiers, les noms étant les divers arguments de la ligne de commande :

```
$ ./compte toto.txt compte.c
toto.txt: 15 caracteres, 4 lignes
compte.c: 245 caracteres, 23 lignes
```

1.4 Exercices à faire... s'il reste du temps

1.4.1 Conversion hexadécimale

Vous avez vu le fonctionnement de la procédure atoi(3) :

```
int atoi(const char *nptr);
```

permettant de convertir une chaîne de caractères représentant un nombre en base 10 en un entier ayant pour valeur ce nombre. Écrivez une procédure permettant de convertir une chaîne de caractères représentant un nombre en base 16 (nombre hexadécimal) en un entier ayant pour valeur ce nombre. Cette procédure doit avoir pour prototype :

```
int ahtoi(const char *nptr);
```

La chaîne à convertir est codée en dur dans le programme et vaut `"A380"`.
Rappel : en base 16, on compte comme suit : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11...

1.4.2 Fichiers : format texte ou binaire

Écrivez un petit programme qui écrit la valeur numérique 12345 dans deux fichiers différents. L'écriture des deux fichiers se fait dans le même programme :

- dans le premier fichier `12345.txt`, cette valeur est à écrire en utilisant la procédure `fprintf(3)` dont le prototype est le suivant :
  ```
  int fprintf(FILE *stream, const char *format, ...);
  ```
- dans le second fichier `12345.bin`, la valeur sera écrite en utilisant la procédure `fwrite(3)` :
  ```
  size_t fwrite(const void *ptr,size_t size,
                size_t nmemb, FILE *stream);
  ```

Dans un terminal, comparez le contenu effectif de ces deux fichiers avec la commande `od(3)` :

```
$ od -c 12345.txt
$ od -c 12345.bin
```

Commentez cette différence.

1.4.3 Format de fichier et manipulation de matrices

Définir un format de fichier (texte) permettant de stocker des matrices numériques non nécessairement carrées de taille maximale `NMAX` (nombre maximal de lignes et nombre maximal de colonnes).

Définir et implémenter des procédures permettant de lire et d'écrire des matrices dans un flot. Les tester avec des fichiers simples (matrices 1x1, 2x2, 2x3).

Écrire un programme permettant de fabriquer des matrices de taille nxm (n et m sont deux paramètres précisés comme arguments du programme), et dont les éléments sont des valeurs entières aléatoires entre 0 et 100.

Générer ainsi une matrice 3x5, puis une matrice 5x4.
Note : un soin particulier doit être apporté au traitement des erreurs lors de la lecture, notamment.

Récupérer le programme `matmult.c` qui effectue la multiplication de deux matrices stockées dans le programme source. Utiliser les procédures mises au point ci-dessus de façon que le programme modifié admette des arguments (2 ou 3) désignant les fichiers contenant les matrices à multiplier :

- ◦ les deux premiers arguments (obligatoires) sont les noms des fichiers contenant les deux matrices à multiplier ;
- ◦ le troisième argument, facultatif, précise le nom du fichier (à créer) contenant la matrice résultat ; si cet argument est absent, la matrice est affichée sur `stdout`.

Tester le programme sur les deux matrices de l'exercice précédent (matrices 3x5 et 5x4).

Chapitre 2

Structures de données en C

2.1 Cours

Ce cours va présenter la notion de *structure de données*, concept permettant de regrouper, de désigner, de manipuler, des données de natures différentes.

2.1.1 Introduction

Jusqu'à présent, nos programmes utilisaient des données simples : caractères, entiers, flottants, et tableaux des précédents. Ce n'était donc que dans l'esprit du programmeur qu'un lien fonctionnel pouvait éventuellement s'établir entre des données de natures différentes. Par exemple, dans le cas des matrices, nous avions été amenés à utiliser deux données :

- une variable qui contient la taille de la matrice ;
- une variable de type tableau à deux dimensions qui contient les valeurs de la matrice.

Lorsque le nombre de données augmente, il devient très problématique de les manipuler sous cette forme « éclatée » : chaque fois que l'on veut manipuler une nouvelle entité, il faut déclarer un nombre de variables important. De plus, si l'on souhaite utiliser une procédure sur l'une de ces entités, il faut lui passer en argument toutes les variables utiles pour le traitement : ceci devient rapidement très lourd, source d'erreurs (si on intervertit deux arguments, par exemple), et potentiellement source de dégradation de performance.

C'est pour résoudre ce type de problème que les langages de haut niveau proposent un mécanisme d'agrégation, permettant de regrouper des données. On désigne en général par *structure de données* un tel objet agrégé. On trouve également la terminologie d'*enregistrement* ou de *record*.

2.1.2 Types structurés en C

2.1.2.1 Définition d'un type structuré

Le langage `C` permet au programmeur de construire ses propres types de données agrégées. Pour cela, le programmeur doit préciser :

- le nom donné au nouveau type ;

◦ sa composition, c'est-à-dire le nom et la nature des données qu'il contient.

Ceci se décrit comme suit :

```
struct nom_structure
{
  type1 var1;
  type2 var2;
  ...
  typen varn;
};
```

qui définit le type `struct nom_structure` (attention : le mot-clé `struct` fait partie intégrante du nom de type) comme étant composé d'une donnée de type `type1` de nom `var1`, d'une donnée de type `type2` de nom `var2` ...

Le nom complémentaire de la structure (ici, `nom_structure`) permet de déclarer ultérieurement des variables de ce type, exactement comme on le fait avec les types de base :

```
struct essai
{
  int toto;
  float bidule;
};

struct essai x;
```

Ce nom de structure doit respecter les mêmes contraintes qu'un nom de variable en C : composé uniquement de lettres, chiffres ou du caractère _, ne commençant pas par un chiffre. Le nombre de caractères autorisés dépend de l'environnement.

Chaque donnée à l'intérieur de la structure prend le nom de *champ* (*field* en anglais). Les contraintes sur un nom de champ sont les mêmes que celles pesant sur un nom de variable.

2.1.2.2 Structures et tableaux, structures imbriquées

On peut tout à fait utiliser un champ de type tableau à l'intérieur d'une structure : ceci se fait de façon similaire à la déclaration d'une variable de type tableau. Voici par exemple une déclaration d'un type matrice, toujours sur le principe de l'utilisation du coin nord-ouest d'une matrice de taille maximale fixée :

```
#define NMAX 32

struct matrice
{
  int n;
  int mat[NMAX][NMAX];
};
```

Une « matrice » au sens défini ci-dessus contient alors la taille réelle de la matrice, et les données de la matrice.

On peut aussi déclarer un champ comme étant d'un type structuré préalablement défini. Par exemple :

```
#define MAX_VAL 32
#define MAX_LIGNE 64

struct vector
{
  int nb;
  double val[MAX_VAL];
};

struct matrix
{
  int nb;
  struct vector v[MAX_LIGNE];
};
```

Une telle structure permet par exemple, de stocker un tableau à deux dimensions, le nombre de lignes étant variable (limité à une valeur maximale, comme d'habitude, ici `MAX_LIGNE`), mais le nombre d'éléments dans chaque ligne étant également variable (limité à la valeur `MAX_VAL`).

À noter également qu'il n'y a aucune ambiguïté sur le champ `nb` de la structure `vector` et le champ `nb` de la structure `matrix`, car ils portent des noms identiques mais dans des structures différentes.

Notons enfin que les deux déclarations de types ci-dessus peuvent être regroupées et donner la version parfaitement identique :

```
struct matrix
{
  int nb;
  struct vector
  {
    int nb;
    double val[MAX_VAL];
  } v[MAX_LIGNE];
};
```

Enfin, un type structuré défini par l'utilisateur étant utilisable tout comme les types de base du langage `C`, on peut déclarer des tableaux de structures :

```
struct matrix tableau_de_matrices[10];
```

Pour résumer, tableau et structures sont deux opérateurs d'agrégation de données que l'on peut combiner autant de fois que désiré.

2.1.2.3 Accès aux champs d'une structure

La désignation du champ `toto` de la variable `x` se fait par la construction du langage (dite *qualification directe*)

```
x.toto
```

Cette notation est utilisable aussi bien en partie droite d'une expression, auquel cas la valeur de l'expression est la valeur contenue dans le champ correspondant de la variable, qu'en partie gauche d'une expression d'affectation, auquel cas le champ est modifié et prend la valeur indiquée en partie droite.

Voici quelques exemples d'utilisation des champs de la variable `x` :

```
...
x.toto=4;
x.bidule=1.1;
for (;x.toto--;)
  x.bidule*=(1.-x.bidule/2.);
(void) printf("x.bidule=%f\n",x.bidule);
...
```

et quelques exemples d'utilisation des champs dans le cas de structures imbriquées :

```
...
struct matrix M;
int i,j;
...
for (i=0;i<M.nb;i++)
{
  (void) printf("Ligne %d:\n",i);
  for (j=0;j<M.v[i].nb;j++)
    (void) printf("\tval[%d]=%f\n",M.v[i].val[j]);
}
```

Noter la succession des qualifications `[]` et `.` dans l'expression de la dernière ligne `M.v[i].val[j]` : `M` est une variable de type `struct matrix`, dont on considère le champ `v`, qui est lui-même un tableau, dont on considère l'élément `i`, qui est une structure contenant les champs `nb` et `val`, ce dernier étant enfin un tableau dont on considère l'élément `j` ! Les qualifications se lisent de « gauche à droite ».

Enfin, dans le cas où l'on dispose de l'adresse d'une structure, par une variable de type adresse de cette structure, par exemple, on peut accéder aux champs directement à partir de l'adresse par la construction (dite *qualification indirecte*) du langage `C ->` :

```
struct matrix M;
struct matrix *addr;
...
```

```
addr = &M;
if (addr->nb > 2)
  ...
```

cette dernière notation étant totalement équivalente à :

```
if ((*addr).nb > 2)
```

2.1.3 Structures et procédures

Une structure peut être utilisée comme argument ou comme résultat d'une procédure.

Cependant, le mécanisme de passage des paramètres et de récupération du résultat propre au langage `C` implique une copie des données : ceci peut être pénalisant dans le cas de structure de taille importante, et on préfère bien souvent utiliser (aussi bien pour les paramètres que pour le résultat) des adresses de structures. Ceci permet également aux procédures appelées, le cas échéant, de modifier le contenu des structures.

Voici par exemple une procédure d'impression d'une matrice définie par une variable de type `struct matrix`, utilisant un passage de l'adresse de la variable :

```
void imprime_matrice (struct matrix *M)
{
int i,j;

  for (i=0;i<M->nb;i++)
  {
    (void) printf("Ligne %d:\n",i);
    for (j=0;j<M->v[i].nb;j++)
      (void) printf("\tval[%d]=%f\n",M->v[i].val[j]);
  }
}
...
struct matrix mat;
...
  imprime_matrice(&mat);
...
```

2.1.4 Structures référençant des structures

Nous avons vu que des structures peuvent contenir d'autres structures. Si l'on reprend l'exemple des matrices, cette imbrication des structures de données correspond bien à l'imbrication « naturelle » de l'objet représenté : une matrice est un ensemble de lignes, une ligne étant un ensemble de valeurs.

Il est cependant des cas où cette imbrication ne peut être satisfaisante. Illustrons-le sur un exemple. On cherche à modéliser informatiquement des personnes : chaque personne a des attributs qui lui sont propres, comme le nom, le prénom, la date de naissance, la taille, le poids...

Il peut cependant être souhaitable de conserver dans la structure de données de la personne les relations familiales comme le conjoint, le père, la mère, les frères et soeurs, les enfants.... Il n'est alors pas possible que chaque personne « contienne » explicitement ces personnes liées car les structures devraient se contenir mutuellement (le père contient le fils, le fils contient le père) !

On pourrait penser à une première solution consistant à numéroter les personnes : par exemple, toutes les personnes sont stockées dans un tableau, et le numéro d'indice dans le tableau est alors un identifiant non ambigu de chaque personne. Mais la seule donnée du numéro ne permet pas d'avoir accès *directement* à la personne concernée : on a besoin d'une procédure de recherche de ce numéro, ce qui peut être pénalisant en termes de performances.

Pour trouver une solution efficace à ce problème, reprenons notre modèle de Von Neuman : chaque donnée dans la mémoire de l'ordinateur est repérée, entre autres, par son adresse, c'est-à-dire le numéro de la (première) case mémoire contenant la donnée. On peut donc dire que cette adresse est un identifiant unique de la donnée ! C'est donc cette adresse que nous allons utiliser, on parle en général de *référence* à la donnée, ou encore de *pointeur* sur la donnée.

On rappelle qu'en C, on peut définir des variables de type adresse sur un type (de base, ou structure, ou même adresse...) par la construction :

```
type_de_donnee_referencee *nom_variable;
```

le caractère * indiquant le fait que c'est une variable de type adresse que l'on définit. On rappelle également que l'on peut désigner l'adresse d'une variable par la construction :

```
&variable
```

Voici quelques exemples de déclarations :

```
/* i est une variable de type int */
int i;
/* addr est une variable de type adresse de int */
/* initialisee a l'adresse de la variable i */
int *addr = &i;
/* tab_addr est un tableau d'adresses de caracteres */
/* soit encore un tableau de chaines de caracteres */
/* tab_addr[0] est (l'adresse de) la chaine "hello" */
/* tab_addr[1] est (l'adresse de) la chaine "world" */
char *tab_addr[10] = {
  "hello","world"
};
```

On rappelle encore que le nom d'un tableau désigne en fait l'adresse du premier élément de ce tableau.

Voici donc une façon de construire une structure de données de type personne, permettant de référencer (éventuellement) un conjoint. Comme l'adresse 0 (macro-définition NULL) est une adresse mémoire dont on est certain qu'elle ne peut contenir une donnée valide, on utilisera cette adresse lorsqu'une personne n'a pas de conjoint.

```
struct person {
  char nom[32];
  char prenom[32];
  float poids;
  float taille;
  struct person *conjoint;
};
```

Voici maintenant une procédure permettant de remplir les champs de la structure (noter que l'on passe l'adresse de la structure à la procédure, car celle-ci doit modifier le contenu de cette structure) :

```
/* procedure de "remplissage" de la structure */
void def_personne (struct person *pers,
  char *nom,char *prenom,
  double poids,double taille,
  struct person *conjoint)
{
  strncpy(pers->nom,nom,31);
  pers->nom[31]=0;
  strncpy(pers->prenom,prenom,31);
  pers->prenom[31]=0;
  pers->poids=(float) poids;
  pers->taille=(float) taille;
  pers->conjoint=conjoint;
}
```

Enfin, voici quelques exemples d'utilisation de cette procédure de remplissage :

```
int main (int argc,char *argv[])
{
struct person alice, bob, oscar;

  /* donnees concernant alice, mariee a bob */
  def_personne(&alice,"dubois","alice",55.,1.68,&bob);
  /* donnees concernant bob, marie a alice */
  def_personne(&bob,"dubois","bob",72.,1.82,&alice);
  /* donnees concernant oscar, sans conjoint */
  def_personne(&oscar,"alone","oscar",85.5,1.95,NULL);
  ...
}
```

2.1.5 Allocation dynamique de structures et tableaux

Nous avons insisté plusieurs fois sur le fait que l'on devait déclarer les variables avant de les utiliser : l'un des rôles de cette déclaration est précisément de permettre au compilateur d'allouer

les zones mémoire nécessaires pour le stockage des données. Ceci impose au programmeur de connaître le nombre de données nécessaires au programme : dans le cas des tableaux et des matrices, nous avons ainsi souvent déclaré des tableaux d'une taille maximale fixée, et utilisé une partie seulement de ce tableau, mais il est totalement impossible, à moins de modifier la valeur de la taille du tableau et donc de recompiler le programme, de traiter des données de taille supérieure !

Il est de nombreux cas où le nombre de données est inconnu a priori, et où il n'est pas non plus possible de réserver une place mémoire maximale fixée. Le langage C offre au programmeur la possibilité d'allouer dynamiquement et à la demande, des emplacements en mémoire pour stocker les données.

2.1.5.1 La procédure malloc

La bibliothèque standard du C fournit une procédure dont le prototype, défini dans le fichier d'en-tête `stdlib.h`, est le suivant :

```
void *malloc (size_t sz);
```

Cette procédure (dont le nom dérive de *memory allocator*) alloue (crée) une zone mémoire susceptible de stocker une donnée d'au moins `sz` octets.

Le résultat de la procédure est une adresse générique de type `void*`, représentant l'adresse mémoire où la procédure a créé la zone mémoire demandée. Le contenu initial de cette zone mémoire est indéterminé (on y retrouve ce qu'il y avait à cet endroit de la mémoire, c'est-à-dire à peu près n'importe quoi !).

Cette valeur peut être stockée dans toute variable de type adresse. Voici un exemple d'utilisation de cette procédure, reprenant le petit programme précédent qui manipule des structures de type personne :

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
struct person
{
  char nom[32];
  char prenom[32];
  float poids;
  float taille;
  struct person *conjoint;
};

/* procedure de "remplissage" de la structure */
void def_personne (struct person *pers,
  char *nom,char *prenom,
  double poids,double taille,
  struct person *conjoint)
```

```
{
  strncpy(pers->nom,nom,31);
  pers->nom[31]=0;
  strncpy(pers->prenom,prenom,31);
  pers->prenom[31]=0;
  pers->poids=(float) poids;
  pers->taille=(float) taille;
  pers->conjoint=conjoint;
}

int main (int argc,char *argv[])
{
struct person *alice, *bob, *oscar;

 alice=(struct person*) malloc(sizeof(struct person));
 bob=(struct person*) malloc(sizeof(struct person));
 oscar=(struct person*) malloc(sizeof(struct person));
 /* donnees concernant alice, mariee a bob */
 def_personne(alice,"dubois","alice",55.,1.68,bob);
 /* donnees concernant bob, marie a alice */
 def_personne(bob,"dubois","bob",72.,1.82,alice);
 /* donnees concernant oscar, sans conjoint */
 def_personne(oscar,"alone","oscar",85.5,1.95,NULL);
 ...
}
```

Plusieurs remarques sur ce programme :

- noter la déclaration de variables de type « adresse de structure personne » au lieu de « structure personne » ;
- l'utilisation de l'opérateur `sizeof` qui permet de connaître la taille en octets d'un type ou d'une variable ; son utilisation permet de réserver la *bonne* taille mémoire pour stocker la structure voulue ;
- la conversion explicite de l'adresse générique rendue par la procédure `malloc`
  ```
  alice=(struct person*) malloc(sizeof(struct person));
  ```
  sans cette conversion explicite, le compilateur peut indiquer un avertissement ; la forme générale d'une conversion explicite de type est :
  ```
  (type_converti) expression_type_initial
  ```
  comme par exemple :
  ```
  (double) 3
  (int) 3.45
  (struct person*) 0
  ```
- l'utilisation dans les appels à la procédure `def_personne` des valeurs contenues dans les variables `alice`, `bob` ou `oscar`, sans `&` devant : on copie les valeurs contenues dans ces variables, qui sont de type adresse ;

- bien noter la distinction entre la variable de type adresse, dont l'emplacement mémoire est réservé par le compilateur, et les objets de type structuré, dont les emplacements mémoire sont alloués dynamiquement ; on ne peut pas d'ailleurs associer dynamiquement un nom de variable à cet emplacement mémoire !
- noter enfin qu'une zone mémoire allouée dynamiquement au sein d'une procédure **reste allouée** même lorsque la procédure est terminée ! Ce comportement est notoirement différent des variables locales à la procédure, dont l'emplacement mémoire est alloué par le compilateur, mais qui sont détruites (emplacement mémoire rendu disponible) lorsque la procédure est terminée.

2.1.5.2 La procédure calloc

On a défini les tableaux comme un arrangement contigu en mémoire d'éléments de type identique. De la même façon que l'on peut allouer dynamiquement des structures, on peut allouer dynamiquement des tableaux : on dispose pour cela d'une procédure nommée `calloc`, dont le prototype est :

```
void *calloc (size_t nb_elems,size_t elem_size);
```

Cette procédure alloue une zone mémoire suffisante pour stocker un tableau contenant `nb_elems` éléments, chaque élément occupant une taille `elem_size`. Comme la procédure `malloc`, cette procédure retourne comme résultat une adresse générique, que l'on peut stocker dans une variable de type tableau, et l'on rappelle ici qu'une variable de type tableau représente en fait l'adresse du premier élément. Une différence notoire par rapport à la procédure `malloc` : **l'espace mémoire ainsi alloué est initialisé à zéro**.

Voici un exemple d'utilisation d'un tableau dont la taille est connue à l'exécution, et dont l'allocation se fait donc dynamiquement :

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

int main (int argc,char *argv[])
{
int N;
int *tab;

 /* la valeur de N (taille du tableau) est le */
 /* premier argument du programme. S'il n'y a */
 /* pas d'argument, on prend la valeur 100 */
 if (argc==1)
   N=100;
 else
   N=atoi(argv[1]);
 /* on alloue un tableau de N elements, chaque */
 /* element est un int (on utilise sizeof) */
```

```
 tab=(int*) calloc(N,sizeof(int));
 if (tab!=NULL)
 {
   (void) printf("tableau tab a l'adresse %p\n",tab);
   return 0;
 }
 perror("calloc");
 return 2;
}
```

Note : on a utilisé dans la procédure `printf` le format `%p`, permettant d'imprimer des expressions de type adresse (constantes, variables, adresses de variables).

Ce programme donne, à l'exécution :

```
$ alloctab 1000
tableau tab a l'adresse 10015008
$ alloctab 300000000
calloc: Resource temporarily unavailable
```

Dans le second cas, la zone mémoire demandée dépasse la mémoire disponible de l'ordinateur, d'où le message d'erreur.

Il est à noter que cette procédure permet d'allouer un tableau à une seule dimension. Il n'est pas possible d'allouer dynamiquement un tableau à p dimensions si les p-1 dernières dimensions ne sont pas connues au moment de la compilation. Par contre, on peut créer un tableau dynamique, dont chaque élément est lui-même un tableau dynamique, comme dans l'exemple ci-dessous :

```
/* creation d'une matrice dynamique nxp */
double **matrice_dynamique (int n,int p)
{
double **mat;
int i,j;

  mat=(double**) calloc(n,sizeof(double*));
  if (mat==NULL)
    return NULL;
  for (i=0;i<n;i++)
  {
    mat[i]=(double*) calloc(p,sizeof(double));
    if (mat[i]==NULL)
      return NULL;
    for (j=0;j<p;j++)
      mat[i][j]=(double) i / (double) j+1;
  }
  return mat;
}
```

Il faut bien comprendre la signification de ce code :

- `mat` est un tableau (une adresse) dont chaque élément est un tableau (une adresse) de valeurs de type `double`, d'où la déclaration `double **mat` (ou de façon équivalente `double *mat[]`);
- on commence par allouer le tableau qui va contenir les lignes, c'est-à-dire `n` tableaux (adresses) ; d'où l'appel initial à `calloc` pour allouer `n` éléments, chacun étant lui-même de type adresse de `double*` ;
- ensuite, on alloue chaque élément du tableau `mat`, qui est un tableau de `p` valeurs, chacune de type `double`, tout ceci étant fait dans la boucle sur `i` ;
- on donne une valeur à chaque élément de la ligne, d'où la seconde boucle sur `j` ;

Noter au passage que l'on utilise pour accéder aux éléments de la matrice l'expression `mat[i][j]`, c'est-à-dire la même expression que pour un tableau à deux dimensions ; le compilateur effectue toutefois un travail différent :

- à partir de l'adresse contenue dans la variable `mat`, il calcule l'adresse de stockage du i-ème élément (`mat[i]`), et récupère la valeur contenue à cette adresse ;
- à partir de valeur précédemment récupérée (qui est une adresse), il calcule l'adresse de stockage du j-ème élément, et récupère la valeur contenue à cette adresse.

Dans le cas d'un tableau à deux dimensions, le compilateur peut directement calculer l'adresse de stockage de l'élément (i, j) à partir de l'adresse du premier élément du tableau.

De la même façon, au point de vue de l'encombrement mémoire, ce type de stockage nécessite, outre les $n * p$ emplacements pour stocker les valeurs, le stockage du tableau des n adresses des lignes (soit $4n$ octets supplémentaires sur un système 32 bits).

2.1.5.3 La procédure free

Ainsi que ceci a été noté, un espace mémoire alloué dynamiquement n'est pas détruit en fin de procédure. Si on veut le libérer, on doit le faire explicitement, par un appel à la procédure `free` dont le prototype est :

```
void free (void *addr);
```

Le paramètre passé en argument à cette procédure est l'adresse (de type adresse générique) d'une zone mémoire préalablement obtenue par un appel à `malloc` ou `calloc`.

Note 1 : de même qu'il est de bonne pratique de fermer un fichier dont le programme n'a plus besoin, il est de bonne pratique de libérer les zones mémoires dont on n'a plus besoin ; en fin de programme, les zones mémoires sont de toutes façon libérées et rendues au système d'exploitation ;

Note 2 : le plus souvent, la mémoire libérée n'est pas **vraiment** rendue disponible pour le système ; elle est à nouveau disponible pour des appels ultérieurs de `malloc` et `calloc` ;

Note 3 : si l'on effectue de nombreux appels à `malloc` et `free`, on peut obtenir une mémoire en « gruyère » (succession de zones libres et de zones allouées) qui peuvent pénaliser les performances. Il existe pour pallier ce problème une version optimisée de la bibliothèque d'allocation dynamique. On peut également trouver des versions de débogage de cette même bibliothèque, qui permettent de trouver plus facilement des bogues se traduisant par un dysfonctionnement du mécanisme d'allocation mémoire ;

Note 4 : que se passe-t-il si on passe en argument à `free` une adresse non obtenue par `malloc` ? Certains manuels précisent :

> Undefined results will occur if some random value is passed as the argument to free.

D'autres sont encore plus alarmistes :

> Absolute chaos is guaranteed if some random value is passed as the argument to free.

2.1.6 Définition de type nommé

On a vu que, pour un type structuré, le mot-clé `struct` (ou `enum` ou `union`, voir plus loin) fait partie intégrante du nom de type. Le langage `C` offre la possibilité de nommer un type défini ; ceci se fait grâce au mot-clé `typedef` (*type definition*), précédant une définition ressemblant à une définition de variable :

```
typedef struct toto
{
  int x;
  float y;
  struct toto *addr;
} T_toto;
```

Dans ce cas, on ne définit pas une variable mais un nom de type (dans l'exemple `T_toto` devient le nom défini pour le type `struct toto`).

On peut d'ailleurs utiliser ce mécanisme pour donner un nom à un type tableau, en utilisant la même syntaxe :

```
typedef int vector[100];
```

On définit ainsi un type « tableau de 100 entiers », au lieu de définir une variable de ce type.

Une fois que l'on a défini un type nommé, on l'utilise exactement comme un type standard du langage :

```
vector V;
T_toto tmp;
...
V[50]=tmp.x;
...
```

Note : le type `FILE` utilisé pour les entrées-sorties est en fait un type nommé défini par un `typedef` dans le fichier d'en-tête `stdio.h` ; on rappelle à ce sujet que les entrées-sorties ne sont pas un élément du langage `C` lui-même, mais font partie d'une bibliothèque standard.

2.2 Compléments de cours

2.2.1 Unions et types énumérés

Outre la construction de structures, le langage `C` offre deux mécanismes similaires : la construction d'unions, et la construction de types énumérés.

2.2.1.1 Énumérations

On a parfois bien envie de représenter des données de nos programmes par des noms symboliques : l'exemple le plus trivial est celui du type booléen, dont nous avons déjà indiqué qu'il n'existait pas dans le langage `C`.

Il existe donc un constructeur particulier du langage permettant de définir un type composé de valeurs prises dans un ensemble, dont les éléments sont définis par un symbole. La déclaration d'un type énuméré fait par la construction :

```
enum nom_de_type { liste de symboles };
```

Les symboles doivent vérifier les mêmes propriétés que les noms de : succession de caractères alphabétiques (minuscules et majuscules), numériques, ou caractère _, ne commençant pas par un chiffre.

Comme pour les structures, le mot-clé `enum` fait partie intégrante du nom de type. Voici un exemple de définition de type « booléen » :

```
enum booleen { vrai, faux };
...
enum booleen condition;
...
condition=vrai;
for (;condition != faux ;)
{
  if (test())
    condition=faux;
}
```

Il faut savoir qu'en interne, tout type énuméré est représenté par un entier : le compilateur se charge de convertir les symboles en valeurs et réciproquement. On peut d'ailleurs imposer les valeurs à certains des symboles en le précisant par une initialisation des symboles dans la déclaration du type énuméré, comme dans l'exemple ci-dessous :

```
enum couleur {rouge,vert=2,bleu=4,orange,jaune=2};
```

Encore une fois, le compilateur se borne à convertir les symboles en valeurs : il n'est dit nulle part que ces symboles représentent forcément des valeurs distinctes, et l'écriture ci-dessus est parfaitement licite !

2.2.1.2 Unions

La construction de structures définie précédemment permet de modéliser des objets définis par une agrégation de données (les champs de la structure). Il est des cas où l'on a besoin de représenter des données de type différents, un seul de ces types de données étant utile pour une instance d'objet donnée.

Prenons l'exemple des variables d'un langage de programmation. Pour chaque variable, on a besoin de stocker son nom (une chaîne de caractères vérifiant certaines contraintes), son type (on se limitera ici à `int`, `double` et chaîne d'au plus 15 caractères), ainsi que sa valeur. On voit que la donnée représentant la valeur de la variable dépend du type.

Une union est un mécanisme de construction permettant justement de définir les différentes représentations d'une donnée, sachant que la place mémoire de chacune des représentations occupe un espace mémoire identique. Pour l'exemple de la variable, on pourrait ainsi définir :

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

#define NOM_MAX 32
#define MAX_CAR 15

struct variable {
  char nom[NOM_MAX];
  enum type_var { entier, flottant, chaine } type;
  union val_var {
    int i;
    double d;
    char str[MAX_CAR+1];
  } val;
};
```

Noter l'utilisation d'un type énuméré pour représenter le type de la variable. Bien distinguer `NOM_MAX` qui est le nombre maximal de caractères pour stocker le nom de la variable, et `MAX_CAR` qui est le nombre maximal de caractères dans le cas où la variable est du type `chaine` !

Voici maintenant une procédure qui affiche les informations contenues dans une structure (noter que, bien que la procédure ne modifie pas le contenu de la structure, on utilise un passage par adresse pour des questions de performance) :

```
void imprime_variable (struct variable *v)
{
  (void) printf("variable %s, type %d, valeur :",
```

```
                  v->nom,v->type);
    switch (v->type)
    {
      case entier:
        printf("%d\n",v->val.i);
        break;
      case flottant:
        printf("%f\n",v->val.d);
        break;
      case chaine:
        printf("%s\n",v->val.str);
        break;
    }
  }
```

Utilisons maintenant cette procédure dans le petit exemple suivant :

```
  int main (int argc,char *argv[])
  {
  struct variable v1,v2,v3;

    strcpy(v1.nom,"i1");
    v1.type=entier;
    v1.val.i=4;
    strcpy(v2.nom,"x0");
    v2.type=flottant;
    v2.val.d=3.141596;
    strcpy(v3.nom,"str");
    v3.type=chaine;
    strcpy(v3.val.str,"Hello world");
    imprime_variable(&v1);
    imprime_variable(&v2);
    imprime_variable(&v3);
  }
```

ce qui donne à l'exécution :

```
  variable i1, type 0, valeur : 4
  variable x0, type 1, valeur : 3.141596
  variable str, type 2, valeur : Hello world
```

On accède aux champs d'une union exactement comme aux champs d'une structure.

Note : il est, encore une fois, de la responsabilité du programmeur de garantir que l'accès au champ utile d'une union est fait à bon escient. Le compilateur ne s'occupe absolument pas de la cohérence des accès. Par exemple, si la variable `v3` utilisée ci-dessus est par erreur déclarée de type `entier`, on obtient le résultat suivant :

```
variable i1, type 0, valeur : 4
variable x0, type 1, valeur : 3.141596
variable str, type 0, valeur : 1214606444
```

Explication : les quatre premiers octets de la chaîne de caractères `'H'`, `'e'`, `'l'` et `'l'` ont été interprétés comme codage sur 4 octets `0x48656c6c` ce qui (sur machine gros-boutiste) correspond à :

$$108 + 256 \times (108 + 256 \times (101 + 256 \times 72)) = 1214606444$$

2.3 Travaux pratiques

2.3.1 Manipulation de fichiers de voitures

On dispose d'un fichier contenant des informations sur des voitures d'occasion : le modèle et le prix. Plus précisément :
- le fichier est de type texte (caractères imprimables du code ASCII) ;
- chaque ligne du fichier correspond à une voiture ;
- chaque ligne du fichier contient le nom du modèle suivi du caractère `;` (point-virgule), suivi du prix (exprimé en euros par un nombre entier ou flottant).

Voici un petit exemple d'un tel fichier :

```
Peugeot 206 CC;7200
Citroen Xsara HDi;2900
Renault Laguna DCi;6300
Citroen Xsara Picasso HDi;7500
```

Le but de l'exercice est d'écrire un programme qui va permettre de trier ce fichier par ordre de prix croissant.

2.3.1.1 Liminaire

Télécharger sur le site Internet du cours le fichier `voitures.c`. Ce fichier contient la définition d'un type `struct voiture`, défini comme suit :

```
/* taille maximale pour un nom de voiture                    */
#define MAXNOM 40

/* une structure de donnees pour stocker une voiture :      */
/* le nom (tableau de MAXNOM caracteres), le prix (double) */
struct voiture {
  char nom[MAXNOM];
  double prix;
};
```

Ce fichier contient également une procédure qui permet de trier par ordre de prix croissant un tableau de structures `struct voiture` (on ne cherchera pas forcément à comprendre cette procédure).

Il contient enfin une procédure principale qui utilise la procédure précédente sur un exemple défini en dur dans le programme : compilez ce programme, testez-le, puis comprenez l'utilisation de la procédure de tri.

2.3.1.2 Procédure de lecture du fichier

Ecrivez une procédure de lecture d'un tableau de voitures à partir d'un fichier. Cette procédure devra avoir le prototype suivant :

```
int lit_fichier_voitures (struct voiture bagnoles[],
                          int nb_voit,
                          char *nom_fich);
```

les trois paramètres représentant respectivement l'adresse mémoire du tableau, le nombre maximal d'éléments de ce tableau, le nom du fichier (une chaîne de caractères).

La procédure devra rendre comme résultat le nombre de voitures lues. Attention : la procédure doit bien entendu tester si ce qui est lu est correct (toute ligne qui contient des données incorrectes doit être ignorée). Faire attention aussi à ne pas lire plus de données qu'il n'y a de place dans le tableau ! Le résultat devra être égal à -1 en cas d'erreur (fichier inexistant ou non lisible).

2.3.1.3 Procédure d'écriture d'un fichier de voitures

Ecrivez maintenant une procédure d'écriture d'un tableau de voitures sur la sortie standard (en utilisant un format identique à celui du fichier original). Cette procédure admet deux paramètres : l'adresse mémoire du tableau et le nombre d'éléments à écrire.

Puis modifiez la procédure précédente afin d'écrire dans un fichier (le fichier original ou un autre fichier), dont le nom sera passé en troisième paramètre de la procédure.

2.3.1.4 Conclusion

Complétez votre programme pour qu'il réalise l'ensemble des traitements voulus. Pour être plus précis, le programme devra considérer que chaque argument qui lui est passé sur la ligne de commande est le nom d'un fichier que le programme doit trier (le résultat du tri devra être écrit dans le fichier original).

Le *status code* du programme sera égal au nombre de fichiers que le programme n'aura pas réussi à traiter (donc 0 en cas de succès).

2.3.2 Gestion de comptes bancaires

Il s'agit maintenant de développer une petite application de gestion simplifiée de comptes bancaires.

On veut que cette application ait le comportement suivant :

- ◦ elle doit présenter une interface utilisateur simple (de type ligne de commande) permettant de réaliser les opérations de base qui sont :
  - ▷ la création d'un nouveau compte ; on attribue un numéro à chaque compte, un titulaire, une adresse, un solde initial nul ;
  - ▷ la consultation du solde d'un compte ;
  - ▷ un dépôt en numéraire sur un compte ;
  - ▷ un retrait en numéraire sur un compte (pas de découvert autorisé !) ;
  - ▷ la consultation des 4 dernières opérations réalisées sur un compte ;
  - ▷ la clôture du compte, qui n'est possible que si le solde du compte est nul (il faut éventuellement effectuer un retrait de ce solde auparavant).
- ◦ elle doit conserver les données saisies sous forme permanente, afin de les retrouver lors d'une prochaine exécution ;
- ◦ l'administrateur de l'agence doit pouvoir accéder au total des soldes des comptes ouverts ;

Afin de se consacrer au plus sur les aspects de modélisation des structures de données, un squelette du programme est fourni, intégrant tous les aspects d'interface utilisateur : menus de consultation, saisie des informations… Ce programme nommé `basebank.c` est disponible sur le site Internet du cours.

2.3.2.1 Liminaire

Récuperer le programme d'exemple, le compiler, le faire tourner afin d'en comprendre le fonctionnement (les différents menus accessibles).

Puis, lire attentivement le code et repérer les procédures qu'il faudra compléter.

2.3.2.2 Analyse et conception

Réfléchir à la modélisation d'une opération, d'un compte bancaire ; réfléchir également à la modélisation de l'agence elle-même.

Réfléchir au problème du stockage permanent des données. Concevoir des mécanismes d'initialisation (récupération des données) et de terminaison (sauvegarde de ces données).

Réfléchir à la façon de réaliser les opérations demandées. Réfléchir de plus à quelques fonctionnalités bien pratiques : retrouver un compte connu par son numéro, retrouver un compte connu par le nom de son titulaire…

2.3.2.3 Implémentation

Implémenter les structures et procédures conçues ci-dessus. Un grand soin doit être apporté, comme d'habitude, au traitement des éventuelles erreurs.

2.3.2.4 Pour conclure, s'il reste du temps…

Réfléchir à l'implémentation en allouant dynamiquement les structures de données.

Chapitre 3

Listes

3.1 Cours

La suite de ce cours va présenter des structures de données classiques, dont on trouve l'utilité dans de nombreux problèmes.

3.1.1 Introduction

Les structures de données qui vont être présentées ne sont que très rarement utilisées telles quelles dans la résolution des problèmes réels. Par contre, les structures réelles sont quasiment tout le temps composées d'éléments pouvant chacun se rapporter à l'une des structures détaillées ci-après.

On retrouve donc une analyse classique :

- ◦ identifier dans un problème des données ou des traitements utilisant des outils de base ;
- ◦ savoir assembler à bon escient ces composants pour résoudre le problème posé.

Ce chapitre s'intéressera à présenter l'une des structures de données fondamentale de l'informatique, la liste, ainsi que quelques uns des algorithmes qui lui sont associés.

3.1.2 Listes chaînées

3.1.2.1 Présentation

On a vu que les tableaux permettaient de représenter en mémoire une collection de données de même nature. Il a également été dit que ces tableaux étaient manipulés sous forme linéaire : la donnée à l'indice $i + 1$ est consécutive en mémoire à la donnée d'indice i.

Il est des cas où cette contiguïté n'est pas satisfaisante. Par exemple, pour ajouter ou supprimer une donnée à un endroit quelconque d'un tableau, on est obligé de *décaler* la *fin* du tableau (les indices supérieurs à l'indice où l'on veut insérer ou détruire la donnée). Si l'endroit de modification du tableau est totalement aléatoire, on obtient une opération dont le coût est, en moyenne, proportionnel à la taille du tableau.

On a alors un problème à résoudre : la notion d'*élément suivant* dans un tableau est une notion implicite. On va donc être obligé de l'expliciter afin de conserver les mêmes techniques

de parcours. Parmi les moyens dont on dispose en langage `C`, on va utiliser l'adresse mémoire afin de désigner de façon unique un élément.

On appelle donc *liste chaînée* ou plus simplement *liste* une structure de donnée constituée d'éléments contenant chacun :

- des données utiles ou une référence (*adresse*, ou *pointeur*) sur des données utiles ;
- une référence (*adresse*, ou *pointeur*) de l'élément suivant.

La liste est elle-même totalement déterminée par la référence (l'adresse) de son premier élément : on nomme souvent ce premier élément *tête de liste*.

Pour indiquer le dernier élément de la liste, on se sert en général d'une adresse dont on est sûr qu'elle ne peut contenir aucune donnée utile : c'est le cas de l'adresse définie par la macro-définition `NULL` dans le fichier d'en-tête `stdio.h`.

L'algorithme 1 propose un exemple de déclaration en `C` d'un type structuré permettant de construire une liste d'entiers.

Algorithme 1 Exemple de réalisation d'une structure de liste

```
struct list_entier {
  int val;
  struct list_entier *suivant;
};
...
struct list_entier *tete=NULL;
```

Insistons encore sur le fait qu'une telle structure ne nous sert que d'exemple ! L'important est de comprendre qu'un élément de liste est une association d'une valeur (ou d'un ensemble de valeurs), avec une désignation (souvent, une *adresse* ou un *pointeur*) qui nous indique où se trouve l'élément « suivant », alors que, dans un tableau, cet élément suivant est implicitement défini comme l'élément dont le numéro est un plus le numéro de l'élément courant. Nous reviendrons naturellement sur ces notions, en explicitant les différences entre ces deux types de structures de données, et leurs avantages et inconvénients respectifs.

3.1.2.2 Procédures de manipulation de listes

Les procédures de manipulation concernent :

- l'initialisation d'une liste (en général, liste vide) ;
- le test, permettant de déterminer si une liste est vide ;
- passage à l'élément suivant ;
- l'ajout d'un élément dans une liste ;
- la suppression d'un élément dans une liste ;
- le parcours de la liste ;
- etc.

Une liste, ou plus précisément, la tête de la liste, sera, dans la suite du chapitre, représenté par un *pointeur* sur le premier élément de la liste (l'adresse mémoire de ce premier élément), et nous

utiliserons par défaut la convention décrite dans l'algorithme 1 page précédente, l'objet `NULL` permettant de représenter une liste vide.

L'initialisation d'une liste est alors triviale : il suffit d'affecter la valeur `NULL` à la variable représentant la tête de la liste.

3.1.2.3 Longueur d'une liste

À titre d'exemple, voici un premier algorithme (algorithme 2) opérant sur nos listes, et qui permet de calculer le nombre d'éléments d'une liste.

Algorithme 2 Taille d'une liste

```
int list_length (struct list_entier *L)
{
int len=0;

    while (L != NULL) {
              L = L->suivant; len++;
    }
    return len;
}
```

Cet algorithme est un parcours de liste typique, utilisant une boucle de type `while`, mettant en œuvre deux opérations élémentaires : le test, déterminant si l'on est arrivé en fin de liste ou non, et le passage au suivant.

3.1.2.4 Impression du contenu d'une liste

L'algorithme présenté ci-dessous (algorithme 3) est trivialement proche du précédent : il s'agit simplement d'imprimer la valeur des éléments rencontrés au cours du parcours de la liste.

Algorithme 3 Impression de liste

```
void list_print (struct list_entier *L)
{
int num;

  for (num = 0 ; L ; num++ , L = L->suivant)
    printf("Element %d = %d\n", num, L->val);
}
```

Note : on a cette fois utilisé une boucle `for`. On rappelle que cette boucle comprend trois parties :

- l'expression d'initialisation (ici `num = 0`) ;
- l'expression de test de boucle (ici `L`, donc valeur « vraie » tant que l'adresse contenue dans `L` est différente de 0) ;
- l'expression de passage à la boucle suivante ; ici, il y a deux expressions (`num++` et `L=L->suivant`), que l'on a réunies en une expression unique en les séparant par des virgules (opérateur dit « d'absorption »).

L'instruction du corps de la boucle est alors réduit à la seule impression.

3.1.2.5 Création d'élément

Nous n'avons pas encore abordé les aspects de création d'élément de liste. Cette opération peut, a priori, se scinder en deux étapes :
- l'obtention d'une zone de mémoire pour la représentation de l'élément ;
- la mise en place de cet élément au sein d'une liste particulière, à l'emplacement désiré.

Comme pour toute réalisation informatique, différentes approches sont possibles pour l'obtention d'une zone de mémoire pour la représentation d'un élément, par exemple :
- déclarer un tableau de structure de `list_entier`, et utiliser les éléments successifs du tableau ;
- utiliser la procédure du système `malloc` pour allouer dynamiquement cette zone.

C'est cette dernière approche que nous utiliserons. Les algorithmes 4 nous permettent respectivement d'allouer et de libérer la mémoire servant à représenter un élément.

Algorithme 4 gestion des éléments

```
struct list_entier *elem_new (int val)
{
struct list_entier *new;

  new = (struct list_entier *)
      malloc (sizeof(struct list_entier));
  new->val = val;
  new->suivant = NULL;
  return new;
}

void elem_free (struct list_entier *elem)
{

  free(elem);
}
```

Fort traditionnellement, nous utilisons comme paramètre de la procédure `malloc` le résultat de l'opération `sizeof`, qui seul nous garantit la portabilité du programme.

L'algorithme propose également une opération de libération de la mémoire associée à un élément. Nous verrons qu'il est possible de changer l'implantation de ces deux opérations, `elem_new` et `elem_free`, sans modifier en rien les autres algorithmes opérant sur des listes.

Important : Enfin, gardons en mémoire le fait qu'*une liste doit être initialisée*, en général en affectant à la variable qui la représente (de type adresse d'élément) la valeur `NULL`.

L'élément étant créé, il convient maintenant de l'introduire dans la liste, à l'emplacement désiré. C'est là qu'il convient de noter l'une des particularités des listes : tous les éléments d'une liste ne sont pas « égaux » en termes de manipulation et de temps d'accès : le premier élément d'une liste est directement accessible au travers du pointeur qui représente la liste ; le second élément n'est accessible que depuis le pointeur qui se trouve dans le premier élément. L'accès à l'élément de rang n nécessite de parcourir les $n - 1$ premiers éléments de la liste ; ce comportement est tout à fait différent de celui des tableaux, dans lesquels le coût d'accès à un élément est indépendant de la taille du tableau et de la position de l'élément dans le tableau. Cet aspect est à prendre en compte lors de l'évaluation d'algorithmes mettant en œuvre des tableaux ou des listes.

Il est relativement simple d'insérer un élément en tête d'une liste. Il suffit que l'ancienne « tête » de liste devienne le suivant de l'élément inséré, et que cet élément devienne la nouvelle tête de liste. L'algorithme 5 réalise cette opération.

Algorithme 5 Insertion en tête

```
/* Inserer un element en tete */

struct list_entier * list_push
  (struct list_entier * L, struct list_entier *elt)
{

  elt->suivant = L;
  return elt;
}

...

tete = list_push(tete,elem_new(12));
```

De même, il est simple d'insérer un nouvel élément après un élément donné. Cette opération est décrite par l'algorithme 6 page suivante.

On peut proposer un algorithme général d'insertion d'un élément en *fin de liste* (on utilise souvent l'expression « *queue de liste* »). Si la liste est vide, l'algorithme insérera l'élément en tête ; si la liste n'est pas vide, l'algorithme insère l'élément après le dernier de la liste.

On notera que cet algorithme impose toujours de réaffecter la variable qui désigne la tête de la liste, puisque cette variable est effectivement modifiée lorsque l'on ajoute un élément à une liste initialement vide.

Algorithme 6 Insertion après un élément

```
/* Inserer un element apres L */

void list_insert
  (struct list_entier *L,struct list_entier *elt)
{

  elt->suivant = L->suivant;
  L->suivant = elt;
}
```

Algorithme 7 Insertion en queue de liste

```
/* Inserer en "queue" de liste */

struct list_entier *list_append
  (struct list_entier *L, struct list_entier * elt)
{
struct list_entier *cur;

  if (L == NULL)
    return list_push(L,elt);
  cur = L;
  while (cur->suivant != NULL)
    cur = cur->suivant;
  list_insert(cur,elt);
  return L;
}

...

tete = list_append(tete,elem_new(30));
```

L'algorithme 7 page précédente nous permet de créer une liste, contenant, dans l'ordre, les éléments 10, 20 et 30, au travers d'une suite d'expressions telle que :

```
struct list_entier * tete;
...
tete = NULL;
tete = list_append(tete,elem_new(10));
tete = list_append(tete,elem_new(20));
tete = list_append(tete,elem_new(30));
...
```

Pourtant, cette écriture n'est pas optimale. La procédure `list_append` nécessite de « traverser » toute la liste pour aller « déposer » un nouvel élément à son extrémité. Pour mettre en place n éléments, il faut ainsi « traverser » 0, 1, puis 2, puis 3, etc, puis $n-1$ éléments, soit au total $\frac{(n-1)n}{2}$ éléments. Le temps (et donc le coût) de création de la liste est donc proportionnel au carré du nombre d'éléments.

Si au contraire, nous utilisons le code suivant pour créer notre liste :

```
struct list_entier *tete;
...
tete = NULL;
tete = list_push(tete, elem_new(30));
tete = list_push(tete, elem_new(20));
tete = list_push(tete, elem_new(10));
...
```

écriture dans laquelle, notons-le bien, nous insérons en tête de la liste les éléments dans l'ordre inverse de celui où nous voulons les voir apparaître dans la liste, nous pouvons acquérir la certitude que la mise en place de chaque élément est d'une durée constante, indépendante de la taille de la liste, et que la création de notre liste est ainsi proportionnelle au nombre d'éléments à mettre en place. Ce code est donc, a priori, meilleur que le précédent.

Comment choisir le « meilleur » algorithme, comment évaluer le coût d'un programme ? Ces questions font l'objet d'une branche particulière de l'algorithmique, qui est la mesure de la complexité. Nous allons développer ci-après les grandes lignes de ce domaine.

3.1.3 Complexité

La *complexité* d'un algorithme est liée à son temps de calcul, à son encombrement mémoire, etc. Il existe habituellement dans tout algorithme un ou plusieurs paramètres qui vont influer directement sur ce temps de calcul ou cet encombrement mémoire. Lorsqu'il s'agit par exemple de calculer la somme d'un ensemble d'éléments, le nombre d'éléments de cet ensemble va clairement influer sur le temps nécessaire pour effectuer ce calcul.

La notation suivante, dite notation en *grand O*, introduite par D.E. Knuth ([3]), définit des classes de fonctions :

$$O(g) = \{f : \aleph \rightarrow \aleph \,/\, \exists c > 0,\, \exists n_0 \in \aleph,\, \forall n \geq n_0, f(n) \leq c \times g(n)\}$$

Identification	Description
$O(1)$	Indépendant du nombre d'éléments
$O(\ln(n))$	Proportionnel au logarithme du nombre d'éléments
$O(n)$	Proportionnel au nombre d'éléments
$O(n\ \ln(n))$	Proportionnel à $n \times \ln(n)$
$O(n^2)$	Proportionnel au carré du nombre d'éléments
$O(n^3)$	Proportionnel au cube du nombre d'éléments
$O(2^n)$	Exponentiel
$O(n!)$	Factoriel

FIGURE 3.1 – Table des complexités des algorithmes

$O(c \times f) = O(f)$
$O(f + g) = O(f) + O(g)$
$O(f \times g) = O(f) \times O(g)$

FIGURE 3.2 – Opérations sur les complexités

Dire qu'une fonction f appartient à la classe $O(g)$, notation dans laquelle g représente une fonction au comportement connu, revient à dire que le comportement de cette fonction f est borné, à un facteur près, par celui de g.

Ainsi, $f(n) \to n \in O(n^2)$, $f(n) \to 431 \times n^2 \in O(n^2)$, mais $f(n) \to n \notin O(\ln n)$. On notera que la première écriture est exacte, mais qu'il est plus précis d'écrire : $f(n) \to n \in O(n)$.

En algorithmique, nous travaillons essentiellement avec des algorithmes dont le comportement dépend de la valeur ou du nombre d'éléments n de l'une des entrées. Si le temps d'exécution d'un algorithme est proportionnel à cette valeur ou à ce nombre d'éléments des données en entrée, nous dirons que l'algorithme est en $O(n)$. On pourra dès lors affirmer, par exemple, que si la taille des données à traiter double, la durée de ce traitement sera également doublée.

Dans la pratique, nous nous intéresserons aux classes de complexités décrites dans le tableau 3.1, page 66. La complexité augmentant, et l'algorithme devenant donc « plus mauvais », au fur et à mesure que l'on descend dans le tableau.

Un peu de terminologie : on parlera d'algorithme *logarithmique* pour désigner un algorithme dont la complexité est en $O(\ln n)$, d'algorithmes *linéaires*, *quadratiques* ou *exponentiels* pour ceux dont la complexité est en $O(n)$, $O(n^2)$ et $O(2^n)$ respectivement.

3.1.4 Évaluation des complexités

Un algorithme se décompose souvent en parties, juxtaposées ou imbriquées, dont il est possible de calculer séparément les complexités. On peut établir aisément les résultats suivants, pour deux fonctions f et g, et une constante c, résumés dans la table 3.2, page 66. Enfin, il convient en général dans une somme de négliger la plus petite des complexités ; nous dirons qu'un al-

Algo.	**10**	**20**	**50**	**80**	**100**
$O(n)$	0.0001	0.0002	0.0005	0.0008	0.001
$O(n \ln(n))$	0.0002	0.00060	0.00196	0.00351	0.00461
$O(n^2)$	0.001	0.004	0.025	0.064	0.1
$O(n^3)$	0.01	0.08	1.25	5.12	10
$O(n^5)$	1	32	52 m	9 h	27 h
$O(2^n)$	0.0102	10.4858	348 ans	4E+09 sièc.	4E+15 sièc.
$O(3^n)$	0.5905	9 h	2E+09 sièc.	5E+23 sièc.	2E+33 sièc.

FIGURE 3.3 – Durées d'exécution de certains algorithmes

gorithme est en $O(n^2)$ si le calcul nous apprend qu'il est en $O(n) + O(n^2)$. Pour nous donner une idée de l'influence de la complexité sur le temps d'un algorithme, le tableau 3.3, page 67, nous donne les durées d'exécution de divers algorithmes, pour diverses tailles des données (de 10 à 100 éléments), la durée d'une opération élémentaire étant la même dans tous les cas (10^{-5} secondes). Ces durées sont exprimées en secondes, sauf indication contraire (*m* pour minutes, *h* pour heure, *ans* et *sièc.* pour siècles si nécessaire). On notera que certaines classes d'algorithmes deviennent vite inutilisables (c'est un euphémisme) lorsque la taille des données augmente.

3.1.5 Complexité des opérations usuelles

Toutes les instructions d'une machine (et, a fortiori, toutes les instructions d'un langage de programmation de haut niveau) ne nécessitent pas la même durée pour s'exécuter. Une élévation à la puissance, par exemple, est « plus lente » qu'une addition. Les différences entre les temps d'exécution individuels peuvent être considérables. Nous considérerons cependant que toutes ces opérations opèrent en un temps borné, indépendant des valeurs qui entrent en jeu dans l'opération. C'est ainsi que toutes les instructions suivantes :

```
a=2+3;
delta=a*x*x+b*x+c;
r=sqrt(eps)*(2-3*abs(p));
if (a>b) a=a-b; else b=a+b/2;
```

sont considérées comme des opérations en $O(1)$, car toutes les variables intervenant dans les expressions sont des scalaires.

En revanche, chaque instruction répétitive devra être analysée avec soin, pour déterminer sa complexité. Ainsi :

```
k=1; while (k<n) k=k*2;
s=0; for (k=1; k<n; k++) s=s+k;
s=0; for (k=1; k<n; k++) for (l=1; l<n;l++) s=s+1/(k+l-1);
```

les trois boucles ci-dessus sont respectivement en $O(\log n)$, $O(n)$ et $O(n^2)$ respectivement. La complexité du programme proposé par l'algorithme 7 page 64, qui décrit la liste pour ajouter un

élément à son extrémité, est clairement $O(n)$. De même, les deux exemples de création de listes qui suivent cet algorithme s'avèrent-ils en $O(n^2)$ et $O(n)$.

3.2 Travaux pratiques

3.2.1 Création de liste

Écrire une procédure `list_alea()` permettant de construire une liste de P éléments, chaque élément étant un nombre entier choisi aléatoirement dans l'intervalle $[0\,; N[$ (N et P sont les deux paramètres de la procédure, le résultat étant l'adresse de la liste construite). Écrire également une procédure de destruction (libération de toutes les zones mémoires des éléments) d'une liste.

Écrire un programme principal testant votre procédure avec comme paramètres $N = 20$ et $P = 10000$.

Puis (après avoir vérifié que le programme fonctionne), modifier le programme en utilisant la procédure `clock(3)` afin de mesurer le temps d'exécution du programme. Tester le temps de calcul pour de grandes valeurs de P (ne pas oublier de supprimer l'affichage de la liste !).

Quelle est la complexité de votre algorithme ?

Note : on pourra réutiliser la procédure écrite au début du cours permettant de générer un nombre entier aléatoire selon une loi uniforme entre deux bornes A et B.

3.2.2 Recherche de nombre

Écrire une procédure `elem_search()` permettant de tester si une valeur (entière) est présente dans une liste ; la valeur et l'adresse de la liste sont les deux paramètres de la procédure, qui fournit un résultat « booléen » (valeur numérique 1 ou 0) selon que la valeur cherchée existe ou non dans la liste.

Écrire un programme qui construit une liste aléatoire de 100 valeurs entre 0 et 9999, puis qui tire des nombres au hasard entre 0 et 9999 jusqu'à en trouver un présent dans la liste.

Quelle est la complexité de votre algorithme ?

3.2.3 Suppression d'un élément

Écrire une procédure `elem_kill()` permettant de supprimer un élément dans une liste (la valeur de l'élément et l'adresse de la liste sont les deux paramètres de la procédure). Si l'élément n'est pas présent, la liste est inchangée ; si l'élément est présent plusieurs fois, on ne supprime que la première occurrence. Le résultat de la procédure est l'adresse de la liste modifiée.

Reprendre le programme de l'exercice précédent, et supprimer l'élément de la liste lorsqu'il a été trouvé. Tester le programme en affichant la liste avant la suppression et la liste après la suppression.

Quelle est la complexité de votre algorithme ?

3.2.4 Création d'une liste triée

Écrire maintenant une procédure qui construit une liste **ordonnée** (par ordre croissant) de P éléments pris aléatoirement dans l'intervalle $[0\,;\,N[$.

NB 1 : la liste doit rester ordonnée à chaque élément ajouté, il ne s'agit pas ici de construire une liste non ordonnée de P éléments que l'on ordonne ensuite !

NB 2 : on ne doit pas ajouter un élément qui est déjà présent dans la liste.

Tester la procédure en écrivant un programme qui construit une liste de 100 valeurs ordonnées entre 0 et 9999, puis qui l'affiche.

Quelle est la complexité de votre algorithme ?

3.2.5 Gestion d'ensemble

On se propose maintenant d'écrire des procédures de manipulation d'ensembles de nombres entiers, que l'on représente par des listes ordonnées. Écrire les procédures suivantes :

- une procédure de création d'un ensemble de P éléments, comportant des entiers aléatoires choisis entre 0 et $N-1$;
- une procédure qui teste si un élément appartient à un ensemble ;
- une procédure qui teste si deux ensembles sont égaux (contiennent les mêmes éléments).

Étudier pour chaque procédure sa complexité ! Essayer, le cas échéant, d'obtenir une complexité optimale. Écrire ensuite les procédures suivantes :

- une procédure de création de l'union de deux ensembles ; au choix, cette procédure, afin de construire son résultat, pourra ou non détruire les deux ensembles initiaux ;
- une procédure de création de l'intersection de deux ensembles ; au choix, cette procédure, afin de construire son résultat, pourra ou non détruire les deux ensemble initiaux.

Tester chaque procédure en l'appliquant à deux ensembles aléatoires (voir procédures précédentes) de 5 éléments entre 0 et 9. Pour chaque procédure, indiquer la complexité de votre algorithme.

Chapitre 4

Listes, files et piles

4.1 Cours

4.1.1 Algorithmique des listes

Les listes, nous l'avons vu, présentent nombre de particularités. Le tableau 4.1 compare quelques complexités d'opérations sur des tableaux et des listes.

Ces différences sont sensibles lorsqu'il faut concevoir des algorithmes bien adaptés aux structures de données qu'ils doivent manipuler. Pour cette raison, nous insisterons à chaque fois sur les complexités des solutions mises en œuvre dans nos exemples.

4.1.1.1 Copie de liste

Recopier une liste consiste à créer une liste similaire à la liste de départ (même taille, mêmes valeurs des éléments), dont les éléments soient physiquement différents de ceux de la liste originelle.

Puisqu'il faut ainsi créer une copie d'élément, voici, décrite dans l'algorithme 8 page suivante, une procédure permettant de copier un élément d'une liste.

On notera dans cette procédure l'expression :

```
* new = * elem;
```

TABLE 4.1 – Tableaux et listes

Caractéristique	Tableau	Liste
Accès à un élément	$O(1)$	$O(n)$
Insertion en tête	$O(n)$	$O(1)$
Balayage	$O(n)$	$O(n)$
Insertion d'élément	$O(n)$	$O(n)$ ou $O(1)$

Algorithme 8 Copie d'élément

```
struct list_entier * elem_copy (struct list_entier * elem)
{

struct list_entier *new;

  new = (struct list_entier *)
          malloc (sizeof(struct list_entier));
  * new = * elem;

  return new;
}
```

dont le but est de recopier le contenu de la structure pointée par la variable `elem` dans la structure pointée par la variable `new`. Cette écriture permet de recopier l'ensemble des champs de la structure.

Cette procédure de recopie d'élément étant définie, on peut proposer des algorithmes de copie de liste . L'algorithme décrit en 9 décrit une première version de copie de liste. Cet

Algorithme 9 Copie(1) de liste

```
struct list_entier * list_copy1 (struct list_entier * li)
{
struct list_entier * cur, * res;

  cur = res = NULL;
  while (li != NULL) {
    cur = li;
    li = li->suivant;
    res = list_append(res, elem_copy(cur));
  }
  return res;
}
```

algorithme est né de l'idée qu'il suffisait de partir d'une liste résultat vide (ici, `res`), et de lui ajouter, en bout, les éléments successifs de la liste initiale.

Cependant, l'on sait que l'ajout en bout de liste est une opération dont le temps d'exécution est proportionnel à la taille de la liste (ici, n éléments au final), et l'on se rend compte que cette opération va être répétée autant de fois qu'il y a d'éléments, n également. La complexité de l'algorithme sera donc $O(n^2)$, ce qui semble, intuitivement, trop coûteux pour une simple opération de recopie !

L'algorithme 10 page ci-contre propose une version dont la programmation est moins évidente, mais dont la complexité est simplement $O(n)$.

Algorithme 10 Copie(2) de liste

```
struct list_entier * list_copy2 (struct list_entier *li)
{
struct list_entier * cur, * res, * elt, * prev;

  cur = res = NULL;
  while (li != NULL) {
    cur = li;
    li = li->suivant;
    elt = elem_copy(cur);
    elt->suivant = NULL;
    if (res == NULL)
      res = elt;
    else
      prev->suivant = elt;
    prev = elt;
  }
    return res;
}
```

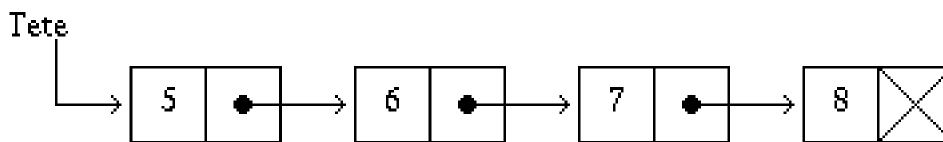

FIGURE 4.1 – Liste initiale

4.1.1.2 Réversion d'une liste

L'algorithme 11 page suivante décrit une procédure de réversion physique d'une liste . Les éléments de la liste sont réarrangés pour constituer une nouvelle liste, dans laquelle ils sont rangés dans l'ordre inverse. On notera qu'après appel de la procédure, la tête de l'ancienne liste pointe sur le dernier élément de la nouvelle liste. Les figures 4.1 et 4.2 page suivante montrent la réversion d'une liste de quatre éléments.

4.1.1.3 Tri de listes

L'une des opérations importantes s'appliquant à des listes (contenant des éléments sur lesquels existe une relation d'ordre) est le tri. On imaginera ici que dans la suite du cours cette notion de tri représente (sauf indication contraire) un tri numérique d'entiers par valeur croissante, mais ceci n'est en rien une restriction sur la généralité des opérations décrites ici.

Algorithme 11 Réversion d'une liste

```
struct list_entier *list_reverse (struct list_entier * li)
{
struct list_entier * cur, * prev;

  cur = prev = NULL;
  while (li != NULL) {
    cur = li;
    li = li->suivant;
    cur->suivant = prev;
    prev = cur;
  }
  return prev;
}
```

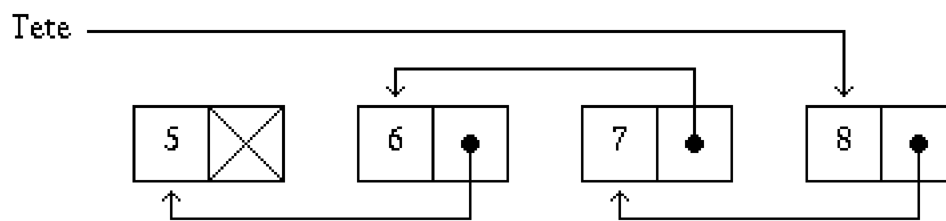

FIGURE 4.2 – Liste retournée

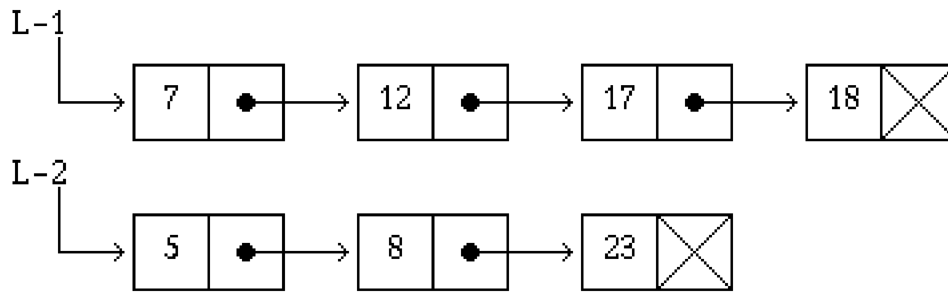

FIGURE 4.3 – Listes avant fusion

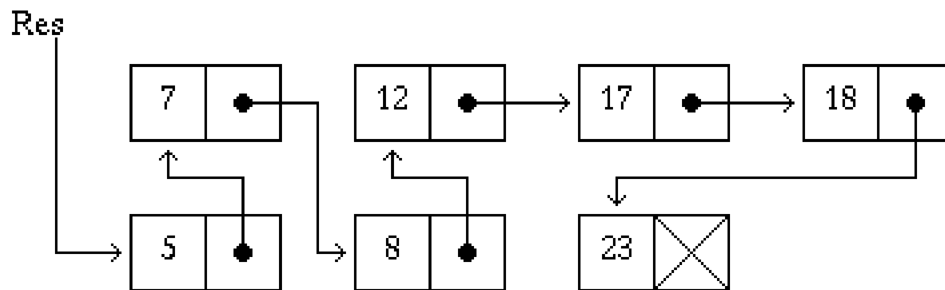

FIGURE 4.4 – Listes après fusion

Nous allons décrire ici un algorithme classique, qui est le *tri par fusion*. Les étapes sont les suivantes :

- l'algorithme prend une liste d'éléments en entrée ;
- si la liste d'entrée est vide, ou comporte un élément unique, elle est donc, de fait, triée, et cette liste est fournie comme résultat ;
- si la liste comporte plus d'un élément, elle est partagée en deux sous-listes ; l'algorithme de partage est lui-même peu important ; il suffit que (à 1 près), les deux sous-listes résultantes aient le même nombre d'éléments ;
- chacune de ces sous-listes est triée (par un appel récursif de la procédure) ;
- les deux sous-listes résultantes (qui sont donc triées) sont fusionnées pour obtenir une liste unique, triée, qui est le résultat de la procédure.

L'opération de fusion consiste à fusionner deux listes triées, pour obtenir une liste résultante qui contiendra les éléments de ces deux listes.

Les figures 4.3 et 4.4 donnent un exemple de la fusion de deux listes telle que décrite ici.

Si l'on raisonne en terme de « profondeur » au niveau de la récursion, l'appel initial de la procédure est l'appel, unique, de niveau 1, sur une liste de taille n. Il effectue lui-même deux appels de niveau 2, sur des listes de taille $n/2$. Il y aura 4 appels de niveau 3, sur des listes de taille $n/4$, huit appels de niveau 4, sur des listes de taille $n/8$, et ainsi de suite. Puisque le

processus s'interrompt dès qu'une liste est de taille inférieur ou égale à 1, il y a, au maximum, $\ln(n)$ niveaux d'appel.

Au niveau 1, la procédure coupe la liste initiale en 2, ce qui fait manipuler n éléments, effectue les deux appels récursifs, puis fusionne les résultats pour obtenir la liste triée finale. Il y a également n élément à fusionner. Ce niveau 1 a donc une complexité en $O(n)$. Le même raisonnement permet de voir que le second niveau consiste en 2 appels de procédures, qui vont elles-mêmes manipuler $n/2$ éléments chacune. La complexité de ce second niveau est également $O(n)$. On peut poursuivre ce même raisonnement jusqu'au dernier niveau, où n procédures s'exécutent, manipulant chacune un seul élément. Pour chaque niveau, la complexité est donc $O(n)$. Puisqu'il y a $\ln(n)$ niveaux, la complexité de l'algorithme est donc $O(n \ln(n))$.

Pour le coin de la culture : on vient d'exhiber un algorithme de tri dont la complexité est $O(n \ln(n))$, ce qui est bien meilleur que le tri par insertion dont la complexité est $O(n^2)$. Peut-on encore faire mieux ? Eh bien non ! Si l'on suppose que la seule opération que l'on peut effectuer est la comparaison de deux éléments (savoir si le premier élément est inférieur, égal ou supérieur au second), alors tout algorithme de tri a une complexité au moins égale à $O(n \ln(n))$. C'est donc un algorithme optimal (ce n'est pas le seul, il existe d'autres algorithmes ayant la même complexité).

4.1.2 Pile

Une *pile* est une structure de données, permettant de gérer (ajouter et retirer) dynamiquement des objets de nature homogène, avec les propriétés suivantes :

- il est possible de tester si une pile est vide ;
- il est possible d'ajouter un objet ;
- il est possible d'extraire un objet d'une pile non vide ; l'objet extrait est celui ajouté le plus récemment à la pile.

L'image d'une série de pièces de monnaies empilées les unes sur les autres donne une bonne idée des propriétés d'une pile.

Les piles sont souvent réalisées au moyen de listes chaînées : « *empiler* » un élément consiste à l'ajouter en tête de la liste, « *dépiler* » à le retirer de la tête de la liste. Ces deux opérations ont une complexité en $O(1)$, et l'allocation dynamique des éléments de la liste satisfait le critère de dynamisme d'une pile.

On dit souvent qu'une pile travaille en *LIFO* (last in, first out), autrement dit « dernier entré, premier sorti ».

4.1.3 File

Une *file* est une structure de données qui permet de gérer dynamiquement (ajouter et retirer) des objets de nature homogène, avec les propriétés suivantes :

- il est possible de tester si une file est vide ;
- il est possible d'ajouter un objet ;

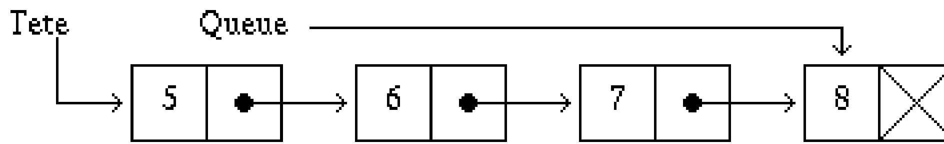

FIGURE 4.5 – File

- il est possible d'extraire un objet d'une file non vide ; l'objet extrait est celui ajouté le plus anciennement à la file.

Les files sont souvent utilisées pour modéliser des files d'attentes, dans lesquelles le premier arrivé est le premier servi. On dit souvent qu'une file travaille en *FIFO* (first in, first out), autrement dit « premier entré, premier sorti ».

Une file est souvent modélisée au moyen d'une liste ; selon la convention choisie :
- on ajoute en tête et l'on retire en queue ;
- on ajoute en queue et l'on retire en tête.

Cependant, ces deux réalisations présentent le défaut que la complexité de l'une des opérations (celle qui fait intervenir la queue de la liste) est en $O(n)$.

Pour cette raison, on associe souvent à la représentation de la liste un pointeur supplémentaire, qui désigne la queue de la liste. La figure 4.5 symbolise la représentation d'une telle file. Grâce à ce pointeur, dont le rôle est de toujours désigner le dernier élément, il est possible d'ajouter un nouvel élément en $O(1)$. La représentation préférée pour les files est donc une liste, avec un pointeur supplémentaire sur la queue de la liste, dans laquelle on ajoute en queue et l'on retire en tête.

4.2 Travaux pratiques

4.2.1 Tri de liste

On se propose d'implanter l'opération de tri par fusion sur des listes, telle que décrite en 4.1.1.3. Écrire et tester une procédure `list_sort`, prenant comme paramètre une liste (en fait, l'adresse du premier élément de cette liste) et rendant comme résultat (l'adresse de) la liste triée. La procédure sera probablement complétée d'une autre procédure, `list_merge`, qui prendra comme paramètres deux listes triées, et fournira comme résultat la fusion des deux listes.

Il est vraisemblable qu'il soit aussi nécessaire d'écrire une procédure `list_split` qui permet de « couper » une liste en deux sous-listes de tailles identiques (à 1 près).

4.2.2 Réalisation de files

On se propose de réaliser une implantation de files de nombres entiers. Pour ceci, on utilisera une liste, dont on décrira les éléments de la façon habituelle. La file elle-même sera décrite au moyen d'une nouvelle structure, que l'on définira, et qui contiendra deux pointeurs sur des

éléments de liste, pointeurs qui désigneront la *tête* et la *queue* de la liste. Les éléments sont représentés par des pointeurs sur des structures, et sont ceux qui sont manipulés par les opération de création et de libération d'éléments (cf. algorithme 4 page 62).

On implantera les quatre opérations suivantes :

`queue_new` création d'une file d'entiers ; procédure sans paramètres, dont le résultat sera l'adresse de la structure représentant la file.

`queue_empty` opération indiquant si une file est vide ou non ; opération prenant comme paramètre l'adresse d'une file (tel que rendu par `queue_new`), et rendant un booléen (c'est-à-dire un entier 0 ou 1).

`queue_insert` opération ajoutant à une file (définie par son adresse, premier paramètre de la procédure) un élément (définie par son adresse, second paramètre de la procédure).

`queue_extract` opération retirant d'une file (premier paramètre de la procédure) l'élément le plus anciennement introduit. Le résultat est l'adresse de l'élément.

On testera ces opérations (en vérifiant le contenu des files au moyen des opérations de manipulation de listes définies au chapitre 3).

Chapitre 5

Graphes

5.1 Cours

5.1.1 Introduction

Un *graphe* est une structure combinatoire qui se compose d'un ensemble de *points*, dits *sommets*, et d'un ensemble d'*arcs*, dits aussi *arêtes*, un *arc* reliant deux sommets.

On peut définir mathématiquement un graphe par un ensemble X et une partie $A \subset X \times X$. Les éléments de X sont les sommets, et l'on dit que l'arête (ou l'arc) (x, y), pour x et y dans X, appartient au graphe si $(x, y) \in A$. On peut également dire qu'il s'agit d'une relation binaire sur X. On notera $G = (X, A)$ le graphe ainsi défini.

En général, les graphes traités informatiquement sont finis, c'est-à-dire que l'ensemble X est fini (ce qui implique que l'ensemble A soit également fini).

Les graphes permettent de représenter nombre de situations fort diverses. Il en résulte que les problèmes de représentation et de recherche en graphes sont nombreux et complexes.

En logistique, par exemple, on peut représenter un réseau de transport (routier, ferroviaire, aérien, fluvial, mixte) par un graphe : les sommets sont les points de chargement ou de déchargement (ainsi que les points de connexion ou d'interconnexion des réseaux), les arêtes représentent les voies de transport.

En théorie des jeux, on peut représenter les « positions » possibles du jeu (les configurations des pions d'un jeu de *solitaire*, par exemple) par un sommet d'un graphe, l'existence d'un arc reliant un sommet A du graphe à un sommet B exprimant le fait que l'on passe de la configuration A à la configuration B en jouant un coup sur le solitaire. Résoudre un problème de solitaire revient à trouver dans ce graphe un chemin passant de la configuration initiale à la configuration finale. Le problème, sachant que le solitaire (anglais) comporte 33 cases pouvant ou non être occupées par un pion est que ce graphe est susceptible d'avoir 2^{33} sommets [1], ainsi qu'un très grand nombre d'arêtes, et n'est pas représentable dans une machine actuelle ; il est encore moins question de le parcourir en des temps raisonnables...

Les graphes auxquels nous nous intéresserons dans ce chapitre sont les graphes :

1. En réalité, beaucoup moins dans le cas où la configuration initiale est celle où l'on a retiré le pion central et où l'on ne représente que les états susceptibles d'être effectivement atteints au cours du jeu.

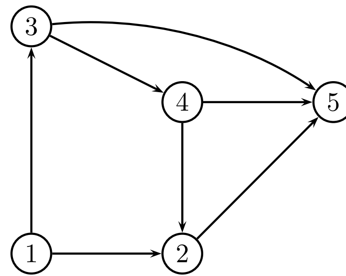

FIGURE 5.1 – Exemple de Graphe

- non orientés, où l'on confond l'arête (x, y) avec l'arête (y, x) (on parle alors d'arc (x, y)) ; on parle également de graphe symétrique ;
- ou graphes orientés, où les arêtes (x, y) et (y, x) sont considérées distinctes ;
- simples, ce qui veut dire qu'il existe au plus une arête d'un sommet A à un sommet B [2].

La figure 5.1 donne une représentation typique d'un graphe orienté.

Enfin, notons le fait, important lors du choix des algorithmes et de la représentation, qu'un graphe peut être *dense* (le nombre d'arêtes issues d'un sommet n'est pas petit devant le nombre n de sommets) ou *clairsemé* (*sparse* en anglais), le cas inverse. Lorsque l'on raisonne en terme de complexité, on peut imaginer que le nombre d'arêtes d'un graphe dense est proportionnel à n^2 et celui d'un graphe clairsemé à n.

5.1.2 Concepts associés

Voici quelques concepts liés aux graphes, et la terminologie associée.

arc liaison orientée (x, y) d'un sommet x à un sommet y d'un graphe orienté.

arête liaison non orientée entre un couple de sommets d'un graphe non orienté.

ordre l'ordre d'un graphe est le nombre de sommets du graphe.

adjacent deux sommets sont adjacents s'il existe un arc joignant ces deux sommets (graphes non orientés). Pour des graphes orientés, on utilise de préférence les termes de **prédécesseur** et de **successeur**.

chemin c'est une suite orientée d'arcs tous différents liant un sommet à un autre sommet d'un graphe orienté.

chaîne suite d'arêtes toutes différentes liant un sommet à un autre sommet d'un graphe non orienté.

distance c'est le nombre minimum d'arcs (ou arêtes) que l'on doit traverser pour passer d'un certain sommet à un autre sommet ; autrement dit, c'est la longueur du plus court chemin liant le sommet de départ au sommet d'arrivée.

circuit c'est un chemin de longueur non nulle allant d'un sommet vers lui-même (dans un graphe orienté).

---

2. Il peut naturellement exister, en outre, une arête allant du sommet B vers le sommet A.

cycle c'est une chaîne de longueur non nulle allant d'un sommet vers lui-même (dans un graphe non orienté). Le fait qu'il existe une arête entre les sommets x et y, et donc que l'on puisse aller de x à lui-même en passant par y ne constitue pas un cycle, puisque cette chaîne utiliserait deux fois la même arête.

acyclique un graphe acyclique ne contient pas de cycle.

connexité un graphe est *connexe* si pour tout couple (A, B) de sommets distincts, il existe au moins un chemin de longueur finie ayant A pour origine et B pour extrémité.

diamètre le diamètre d'un graphe est la plus longue des distances entre tous les couples possibles de sommets.

degré le degré d'un sommet est le nombre d'arcs arrivant à ce sommet (degré entrant) ou partant de ce sommet (degré sortant). Dans un graphe non orienté, le degré d'un sommet est le nombre d'arêtes issues de ce sommet.

valué un graphe est dit valué si des valeurs numériques sont associées aux arêtes. Ces valeurs sont en principe positives. Le coût d'un chemin peut ainsi être calculé comme la somme des valeurs des arcs le composant.

5.1.3 Représentation des graphes

Il existe deux représentations classiques pour les graphes, dites par *matrice d'adjacence*, et par *liste d'adjacence*.

5.1.3.1 Matrice d'adjacence

Dans cette représentation, le graphe est représenté par une matrice booléenne (i.e. ne contenant que des 0 et des 1) M de taille $N \times N$, N étant le nombre de sommets du graphe. Les sommets sont numérotés (par exemple de 0 à $N-1$), et l'élément $M[i, j]$ indique, par sa valeur (1 ou 0) s'il existe ou non un arc du sommet i vers le sommet j.

Quel que soit le nombre d'arcs, la représentation par matrice booléenne occupe, en choisissant la représentation la plus compacte, N^2 bits, soit $\frac{N^2}{8}$ octets.

5.1.3.2 Liste d'adjacence

Dans cette représentation, on conserve pour chaque sommet une liste des sommets qui lui sont adjacents. En `C`, on peut choisir de représenter les sommets, numérotés de 0 à $N-1$, par un tableau de N pointeurs sur les listes d'adjacence.

Dans une telle liste, on peut utiliser une structure pour chaque arc, dans laquelle on représente le sommet vers lequel mène l'arc, par exemple par un entier représentant le numéro de ce sommet, et bien sûr un pointeur vers le sommet adjacent suivant. Une telle représentation nécessite typiquement 8 octets par arc (4 octets pour représenter le numéro du sommet, 4 octets pour représenter le pointeur vers le suivant). Si N est le nombre de sommets, et A le nombre d'arcs, la taille en mémoire est au minimum de $4N + 8A$ octets.

Si le graphe à représenter est fixe (n'a pas à évoluer au cours du calcul), on peut représenter les sommets adjacents à un sommet donné par les éléments successifs d'un tableau, suivi d'un

marqueur de fin. Si le graphe a moins de 2^{16} sommets, ces éléments peuvent être des entiers 16 bits, soit 2 octets par arc, plus le marqueur de fin, 2 octets par sommet. La taille mémoire est alors $6N + 2A$ octets.

5.1.3.3 Comparaison des représentations

Tailles des représentations : La représentation par liste, pour des graphes de petites tailles ou denses, est coûteuse devant la représentation par matrice. Au contraire, si le graphe a un nombre de sommets important, et est relativement clairsemé, la représentation par liste d'adjacence est plus intéressante.

Coût de l'utilisation : Il est un peu coûteux (à cause des manipulations d'extraction de bits) d'aller retrouver l'élément $M[i, j]$ dans une matrice booléenne ; cependant un tel accès est en $O(1)$. Savoir s'il existe un chemin entre i et j avec une représentation par listes nécessite de parcourir la liste d'adjacence du sommet i. Si le graphe est clairsemé, le nombre d'arcs issus d'un sommet est très faible, et l'on peut considérer que cette recherche est en $O(1)$. Si ce n'est pas le cas (graphe dense), la longueur de la liste peut être considérée comme proportionnelle au nombre de sommets, et la complexité de cette opération sera évaluée en $O(n)$.

Inversement, si un algorithme a besoin de trouver tous les arcs issus d'un sommet, cette recherche est en $O(n)$ pour la représentation par matrice (il faut balayer toute la ligne de la matrice), mais en $O(1)$ dans le cas de la représentation par listes pour un graphe très clairsemé.

Le problème inverse, trouver les chemins qui mènent à un sommet donné est simple pour la représentation par matrice ($O(n)$), mais plus coûteux pour la représentation par listes pour des graphes denses ($O(n^2)$).

Autres aspects : Si le graphe doit évoluer au cours du temps, il est clair que la représentation par matrice permet d'ajouter ou de supprimer aisément des arcs. En revanche, ajouter ou supprimer des sommets nécessite une recopie de la matrice, opération qui est en $O(n^2)$. La représentation par listes se prête mieux à l'ajout ou la suppression de sommets, bien qu'il soit alors nécessaire de choisir des représentations plus complexes, donc plus coûteuses, si l'on veut pouvoir effectuer de manière performante ces opérations.

Choix : Le choix de la « meilleure » représentation dépend donc de nombre de paramètres, qui sont liés certes à l'aspect du graphe qui doit être manipulé, mais également à nombre d'autres considérations qui peuvent être liées aux algorithmes à appliquer, ou encore aux procédures dont on dispose.

5.1.4 Problèmes sur graphes

La théorie des graphes s'intéresse à nombre de problèmes abstraits, qui à leur tour représentent des situations et des problèmes tout-à-fait concrets. Voici quelques exemples de problèmes sur graphes.

5.1.4.1 Parcours de graphe

On s'intéresse à énumérer tous les sommets accessibles depuis un sommet donné. Ce parcours peut s'effectuer en profondeur (on va le plus vite possible le plus loin possible du sommet de départ) ou en largeur (il convient d'énumérer en premier tous les sommets à une distance 1 du sommet de départ, puis ceux qui sont situés à une distance deux, etc).

Ce problème d'énumération est extrêmement fréquent : recherche de chemin, de plus court chemin, jeux, etc.

5.1.4.2 Fermeture transitive

La *fermeture transitive* d'un graphe $G = (X, A)$ est la relation transitive minimale contenant la relation (X, A). Il s'agit d'un graphe $G* = (X, A*)$, tel que $(x, y) \in A*$ si et seulement s'il existe un chemin dans G d'origine x et d'extrémité y.

Ce problème se résout typiquement au moyen d'un produit booléen de matrices. Le calcul de $G*$ s'effectue par itération d'une opération qui ajoute à A les arcs (x, y) tels qu'il existe un z qui a x pour prédécesseur et y pour successeur.

5.1.4.3 Arbre de recouvrement

Un *arbre de recouvrement,* ou *arbre recouvrant* est un *graphe partiel*, sous-ensemble d'un graphe non orienté connexe, possédant l'une des propriétés suivantes (équivalentes) :

- le graphe est connexe, mais cesse de l'être dès que l'on supprime l'une des arêtes ;
- le graphe est connexe et possède n sommets et $n - 1$ arêtes ;
- il existe une chaîne et une seule joignant deux sommets quelconques ;
- le graphe possède n sommets et $n - 1$ arcs, et aucun cycle.

Les figures 5.2 et 5.3 page suivante montrent un graphe connexe et l'un des arbres de recouvrement associés.

Un exemple typique d'utilisation d'un arbre de recouvrement est celui de l'alimentation en eau d'un bâtiment, ou d'un quartier. Les sommets du graphe représentent les points d'eau à alimenter, les arcs les endroits où il est possible de faire passer des tuyaux. Calculer l'arbre de recouvrement associé au graphe permet de trouver une solution au problème de l'alimentation des points d'eau. Dans ce type de problème, on utilise souvent des graphes valués, c'est-à-dire que l'on associe des coûts aux arcs (par exemple, la longueur physique de l'arc), et on cherche un arbre de recouvrement de coût minimal.

5.1.4.4 Circuits

On recherche la présence de circuits (cycles) dans un graphe, c'est-à-dire de chemins de longueur non nulle liant un sommet à lui-même.

Une application pratique est celle de la planification de la réalisation d'un projet complexe, par exemple la construction d'un bâtiment. Cette réalisation fait appel à nombre d'opérations différentes (ex : construction des murs, pose de la plomberie, de l'électricité), opérations qui sont éventuellement dépendantes les unes des autres par des contraintes d'antériorité : ainsi, on ne peut guère envisager de poser la plomberie avant d'avoir achevé les murs. Chaque opération peut

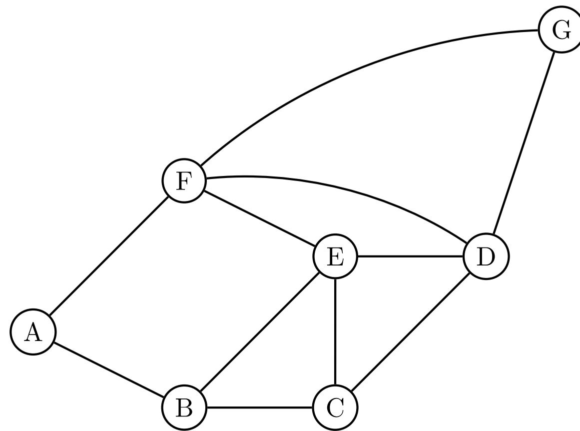

FIGURE 5.2 – Exemple de Graphe connexe

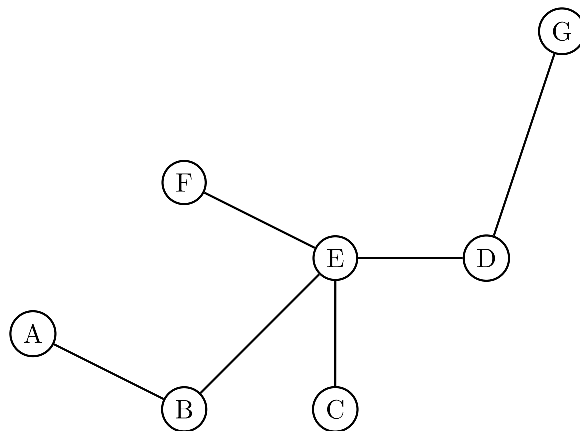

FIGURE 5.3 – Un arbre de recouvrement du graphe 5.2

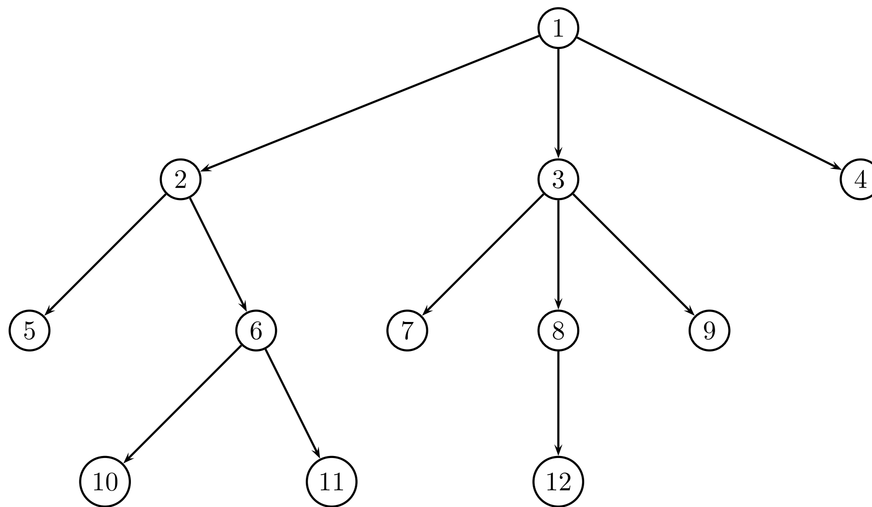

FIGURE 5.4 – Exemple d'arbre

être représentée par le sommet d'un graphe, un arc allant de A vers B indiquant que l'opération A doit être achevée avant que l'opération B puisse débuter. Le graphe obtenu doit être acyclique ; dans le cas contraire, c'est qu'il existe un problème grave dans la réalisation…

Rappel : sauf dans certains algorithmes (liés par exemple aux chaînes de *Markov*), on considère qu'il n'existe pas d'arc joignant un sommet à lui-même.

5.1.5 Arbres

Les *arborescences*, souvent appelées improprement *arbres* constituent un sous-ensemble des graphes [3]. Un arbre est un graphe orienté connexe, ayant les propriétés suivantes :

- il existe un sommet unique, dit *racine* de l'arbre, n'ayant aucun prédécesseur ;
- chaque sommet autre que la racine a un prédécesseur unique ;
- pour chaque sommet, il existe un chemin unique menant de la racine à ce sommet.

Les arbres sont souvent représentés par un schéma dans lequel on place la racine en haut ou à gauche (cf. figure 5.4).

Bien que les algorithme relatifs aux graphes puissent s'appliquer aux arbres, il existe souvent, compte tenu des propriétés particulières des arbres, des algorithmes qui leur sont spécifiques [4]. Ainsi, chaque sommet ayant au plus un prédécesseur, un arbre à n sommets peut être représenté par un vecteur de n éléments, associant à chaque sommet son prédécesseur.

3. On considère qu'un arbre est un graphe non orienté, comprenant des sommets et des arêtes (cf. par exemple les « arbres de recouvrement », 5.1.4.3), et qu'une arborescence est un graphe orienté, constitué de sommets et d'arcs.

4. On peut de même considérer que les listes constituent un sous-ensemble des arbres, donc des graphes, mais algorithmes sur listes sont naturellement mieux adaptés que des algorithmes plus généraux s'appliquant aux arbres !

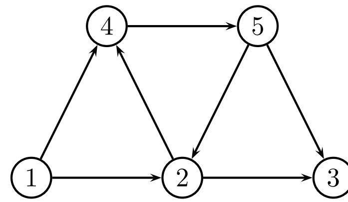

FIGURE 5.5 – Exemple de graphe

5.2 Travaux pratiques

5.2.1 Représentation de graphes

On se proposer de construire un ensemble de procédures permettant la représentation et la manipulation de graphes. On se limitera à des graphes de 256 sommets au maximum. La représentation adoptée sera celle d'une matrice d'adjacence, avec les conventions suivantes :

- un graphe sera représenté par une structure (`struct graph`) contenant l'ordre du graphe (un entier), et une matrice de caractères[5] `m` de taille 256 par 256.
- on utilisera chaque caractère pour représenter un arc, la valeur 1 de `m[i][j]` indiquant qu'il existe un arc reliant `i` à `j`, la valeur 0 qu'il n'en existe pas.
- les sommets seront numérotés de 1 à N.

On écrira les procédures suivantes :

graph_new procédure qui rend l'adresse d'une structure nouvellement allouée.

```
struct graph * graph_new (int ordre);
```

Le graphe ainsi créé ne contiendra aucun arc.

graph_print procédure qui imprime une vision (avec des 0 et des 1) de la matrice d'adjacence du graphe passé en paramètre (sous la forme d'une adresse).

```
void graph_print (struct graph *);
```

graph_arc procédure indiquant s'il existe un arc menant de i à j. Les nombre i et j sont des entiers compris entre 1 et n, ordre du graphe.

```
int graph_arc (struct graph *g, int i, int j);
```

graph_arc_set procédure créant ou détruisant l'arc menant de i à j selon que k vaut 1 ou 0.

```
void graph_arc (struct graph *g, int i, int j, int k);
```

À partir de ce premier ensemble de procédures, on écrira une procédure permettant de lire la représentation d'un graphe depuis un fichier texte. Cette représentation se composera d'une première entrée indiquant l'ordre du graphe, puis d'une entrée par sommet [numérotés de 1 à N] indiquant les numéros de sommets vers lesquels il existe un arc depuis le sommet courant.

5. Cette convention est moins efficace en termes d'encombrement de la mémoire, puisque l'on utilise 8 bits pour représenter l'existence d'un chemin, alors qu'en principe un bit suffit ; elle simplifie cependant grandement la programmation, et rend un peu plus efficaces les procédures de manipulation du graphe.

Les entrées seront séparées les unes des autres par le caractère « ; ». Des blancs et passages à la ligne pourront être insérés si désiré. Ainsi, le graphe de la figure 5.5 page ci-contre sera représenté par un fichier ayant le contenu suivant :

```
5

2 4;3 4;;5;2 3
```

On notera qu'une entrée vide dans un tel fichier indique qu'il n'existe aucun arc issu du sommet correspondant.

5.2.2 Recherche de circuit

On se propose d'implanter un algorithme de recherche de circuit dans un graphe orienté. Pour ceci, on peut utiliser l'algorithme suivant :

- partir de l'ensemble X des sommets et A des arcs ;
- répéter l'opération suivante tant qu'il reste des sommets dans X ;
  - rechercher les sommets qui n'ont pas de prédécesseurs, les retirer de X.
  - retirer de A les arcs issus de ces sommets

S'il reste des sommets dans X, mais qu'il n'y a plus de sommets sans prédécesseur [6], c'est qu'il existe au moins un circuit dans G.

Écrire la procédure qui détermine s'il existe des circuits dans un graphe donné en paramètre.

5.2.3 Calcul de connexité

On se propose de concevoir un algorithme permettant de découvrir, à partir de la matrice d'adjacence d'un graphe, s'il existe des chemins de longueurs 2, puis 3, 4, 5, etc dans le graphe.

En déduire le calcul de la fermeture transitive du graphe.

Quel est la complexité des algorithmes ainsi réalisés ?

6. De tels sommets sans prédécesseur sont dits *sources* ; de même, les sommets sans successeurs sont dits *puits* (anglais *sink*).

Chapitre 6

Algorithmes sur graphes

6.1 Cours

6.1.1 Parcours de graphe

Dans nombre de problèmes, le parcours de graphe revêt une importance essentielle. On peut par exemple désirer trouver l'ensemble des sommets accessibles depuis un sommet donné (l'exercice 5.2.3 page 87 fournit un algorithme permettant d'effectuer ce calcul, mais sa complexité est élevée), ou encore énumérer l'ensemble des sommets accessibles depuis un sommet donné, etc.

6.1.1.1 Parcours en profondeur

Nous allons décrire ici un parcours de graphe en *profondeur*. Cet algorithme sert de support à la résolution d'une grande classe de problèmes, qui peuvent être liés aussi bien aux arbres qu'aux graphes.

Un parcours en profondeur consiste à partir d'un sommet donné d'un graphe (ou de la racine, dans le cas d'un arbre), et à énumérer ses successeurs, puis, choisissant l'un de ces successeurs, à énumérer les successeurs de celui-ci, etc. C'est l'opération de « descente » dans le graphe. Lorsqu'il n'est pas possible d'aller de l'avant, on revient en arrière, on choisit un autre successeur, et l'on effectue une nouvelle descente, jusqu'à ce que tous les chemins possibles issus du sommet de départ aient été énumérés.

On effectuera la sélection des successeurs ou le test d'arrêt en fonction des besoins. En particulier, si le graphe comporte des circuits, certains chemins peuvent être de longueur infinie ; il convient alors, en fonction des besoins de l'algorithme, de prendre les précautions nécessaires pour éviter ce cas de figure, par exemple en veillant à ce que chaque sommet soit utilisé une fois et une seule au cours de la descente. Ce problème ne se pose pas dans le cas des arbres, où il n'y a pas de circuit et où chaque noeud est accessible par un chemin et un seul depuis la racine.

Notons enfin que dans nos descriptions d'algorithmes, nous utilisons un ensemble de procédures dont les spécifications sont décrites dans l'exercice 5.2.1 page 86, et qui sont accessibles sur le site :

`http://kiwi.emse.fr/POLE/courssda.html`

6.1.1.2 Choix des structures de données

Si l'on part du principe qu'un sommet ne sera exploré qu'une fois au cours de la descente, un tableau `pil` de taille n, n étant l'ordre du graphe, suffit à conserver la liste des sommets explorés. À chaque sommet, il convient d'associer la liste des successeurs qu'il reste à explorer, soit, au maximum, n entrées pour chaque sommet. Une matrice `pos` de taille $n \times n$ convient pour représenter cette structure. Une variable `pt` indique le niveau atteint. Enfin, s est le sommet de départ de notre recherche.

6.1.1.3 Algorithme général de parcours

Voici une première ébauche de l'algorithme. Commençons par quelques structures de données :

```
int pos[NMAX][NMAX];
int pil[NMAX];
int pt,succ;
int i, j, c;
```

L'algorithme démarre avec le sommet de départ, s. On le place dans la pile. On utilise un indicateur, $succ$ (comme « succès », ou encore « il existe un successeur »), initialement positionné à 1 puisque l'on va explorer les successeurs du sommet s.

```
pt = 0;
pil[pt] = s;
succ = 1;
```

L'algorithme consister à « boucler » sur la descente. Au cours de celle-ci, on empile les sommets rencontrés. Si l'on ne peut pas aller plus loin, on va dépiler, pour explorer d'autres successeurs du sommet parent. Une fois tous les trajets essayés depuis s, l'algorithme va décrémenter pt, le rendant négatif ; d'où le test de poursuite de la boucle.

```
while (pt>=0) {
  if (succ != 0) {
```

La pile contient un sommet courant c, dont on va chercher les successeurs. Tel que programmé ci-dessous, l'algorithme conserve tous les successeurs du sommet c. Il faut naturellement être plus restrictif, en fonction du but recherché, pour décider si tous les successeurs conviennent ou non ; en particulier, comme expliqué ci-dessus, on peut décider de ne retenir que les sommets qui n'ont pas encore été rencontrés au cours de cette descente.

```
    c = pil[pt];
    i = 0;
    for (j=1; j<=n; j++) {
      if (graph_arc(g,c,j)) {
        /* Decider ou non si le sommet convient */
        pos[pt][i] = j; i++;
```

```
      }
    }
    for ( ; i<n; i++)
      pos[pt][i] = 0;

  }
```

On va maintenant choisir le premier successeur disponible au niveau courant. Si l'on en trouve un ($succ$ est alors différent de 0), on va l'empiler, puis continuer la boucle pour chercher ses propres successeurs, non sans l'avoir retiré de la liste des successeurs à explorer.

```
  for (i=0; i<n && ((succ=pos[pt][i])==0); i++) {
  }
  if (succ != 0) {
    pos[pt][i] = 0; /* eliminer succ */
    pt++;
    pil[pt]=succ;
  }
```

Si l'on a épuisé tous les successeurs de ce niveau, on revient au parent en décrémentant le compteur *pt*. Comme on l'a signalé, le test de la boucle générale permet d'arrêter l'algorithme lorsque *pt* devient négatif.

```
  else {
    pt--;
  }
}
/* Fin */
```

6.1.1.4 Exemple : recherche de chemin

Imaginons que notre algorithme consiste à chercher les chemins entre s et d, sommet de destination. Nous considérerons que pour un sommet intermédiaire k, un successeur est à étudier s'il est différent de d, et s'il n'a pas déjà été atteint au cours de la descente, autrement dit s'il n'est pas dans les $pt + 1$ premiers éléments de `pil`. Si l'un des successeurs est d précisément, nous avons trouvé un chemin de s à d, qui passe par les sommets dont les numéros sont dans les $pt + 1$ premiers éléments de `pil`. Voici donc la version appropriée du choix des successeurs :

```
if (graph_arc(g,c,j)) {
  int p;
  if (j == d) {
    /* on a trouve un chemin ; on l'imprime */
    for (p=0; p<=pt; p++)
      printf("%d ", pil[p]);
    printf("%d\n", d);
  }
```

```
    else {
      int incl = 1;
      /* Sommet deja trouve ? */
      for (p=0; p<=pt; p++)
        if (pil[p] == j)
          incl = 0;
      if (incl) {
        pos[pt][i] = j; i++;
      }
    }
  }
```

L'algorithme ci-dessus imprime tous les chemins trouvés entre s et d. Il serait possible, lorsqu'un chemin est rencontré, de ne conserver que le plus court, et de l'imprimer à la fin.

6.1.1.5 Exemple : sommets accessibles depuis un sommet donné

Si notre but est de construire la liste des sommets accessibles depuis un sommet donné, nous pouvons utiliser un tableau à n entrées, `vus`, qui indiquera pour chaque sommet s'il a déjà été rencontré dans l'exploration.

```
int vus[NMAX+1];
for (i=0; i<=n; i++)
  vus[i] = 0;
vus[s] = 1;
```

La procédure de sélection des successeurs consiste simplement à vérifier que le sommet n'a encore été jamais rencontré au cours de toute l'exploration de l'arbre ; si le sommet a déjà été rencontré, ses descendants ont été explorés (ou vont l'être), et il n'est pas utile de le faire à nouveau. L'opération de sélection des descendants s'écrit alors :

```
if (graph_arc(g,c,j) && (!vus[j])) {
  pos[pt][i] = j; vus[j]=1; i++;
}
```

Lorsque l'exploration du graphe est achevé, on peut par exemple imprimer la liste des sommets rencontrés, et donc accessibles depuis le sommet s :

```
/* Imprimer la liste des sommets accessibles */
for (i=1; i<=n; i++)
  if (vus[i])
      printf("%d ", i);
print("\n");
```

Comme le montre ce programme, la différence avec l'algorithme précédent réside uniquement dans le choix des successeurs à explorer ; ici un sommet est conservé seulement s'il n'a encore jamais été rencontré au cours de l'algorithme.

Ce même algorithme pourrait être modifié aisément afin de construire un arbre de recouvrement du graphe.

6.2 Travaux pratiques

6.2.1 Fermeture transitive

On se propose de construire la *fermeture transitive* d'un graphe G, c'est-à-dire le graphe G' dans lequel un arc entre i et j indique qu'il existe dans G un chemin entre i et j.

L'exercice 5.2.3 page 87 proposait une première méthode de calcul de la fermeture transitive d'un graphe, à travers une séquence de produit booléen de matrice (cf. 5.1.4.2 page 83). Cet algorithme présente l'inconvénient d'être en $O(n^4)$.

On se propose d'implémenter l'algorithme de Roy-Warshall pour la recherche de la fermeture transitive d'un graphe représenté par sa matrice d'incidence M, qui consiste à, pour tout k tel que $M[i, k]$ et $M[k, j]$, ajouter l'arc (i, j) à la matrice.

On implémentera l'algorithme, et on calculera sa complexité.

6.2.2 Recherche de circuit

On se propose de rechercher la présence de circuits dans un graphe G.

L'opération de fermeture transitive permet de détecter l'existence de tels circuits, puisque s'il existe de 1 sur la diagonale de la matrice représentant le graphe de la fermeture transitive de G, ceci indique qu'il existe un chemin allant d'un sommet à lui-même, et donc un circuit.

On peut cependant proposer un algorithme en général plus performant, qui est le suivant :

- ◦ partir de l'ensemble X des sommets et A des arcs ;
- ◦ répéter l'opération suivante tant qu'il reste des sommets dans X ;
  - ▷ rechercher les sommets qui n'ont pas de prédécesseurs, les retirer de X.
  - ▷ retirer de A les arcs issus de ces sommets

S'il reste des sommets dans X, mais qu'il n'y a plus de sommets sans prédécesseur [1], c'est qu'il existe au moins un circuit dans G.

Écrire la procédure qui détermine s'il existe des circuits dans un graphe donné en paramètre.

1. De tels sommets sans prédécesseur sont dits *sources* ; de même, les sommets sans successeurs sont dits *puits* (anglais *sink*).

Deuxième partie

Annexes

Annexe A

Modularité des programmes en C

Avant-propos

Le but de ce document est de présenter comment on peut développer des programmes volumineux en C (plusieurs milliers de lignes) en utilisant un principe de découpage (fonctionnel) en modules indépendants du point de vue de l'implémentation.

La première partie détaille les différentes étapes de la construction d'un programme exécutable à partir d'un source en C. La seconde partie indiquera comment on peut utiliser ce mécanisme pour décomposer un programme en modules.

A.1 Construction d'un programme C

Il n'est pas inutile de revenir à la façon dont nous construisons nos programmes simples. Construire un programme C, c'est, *a minima*, écrire un fichier source (nommons-le `toto.c`) qui contient une procédure nommée `main`. Voici un exemple d'un tel fichier :

```
#include <stdio.h>

int main (int argc,char *argv[])
{
  puts("Hello world");
  return 0;
}
```

Considérons la première ligne, qui contient la directive d'inclusion du fichier d'en-tête `stdio.h` : on a expliqué que cette directive permettait au programme (au compilateur, plus précisément) d'être informé de l'existence d'une procédure nommée `puts`, admettant un premier paramètre de type `char*` (adresse mémoire d'un caractère, ou plus exactement, chaîne de caractères), admettant éventuellement des paramètres supplémentaires, et retournant un résultat de type entier. Détaillons un peu ce processus.

A.1.1 Le préprocesseur

Le compilateur gcc comme tous les programmes du même type est en fait un « enchaîneur de passes » : son rôle est, à partir d'un fichier source (qui peut être un source C mais aussi Fortran, C++...) de produire un programme exécutable en appelant plusieurs programmes successifs (l'enchaîneur de passes stoppe son traitement si l'une des phases échoue).

La première de ces passes est le préprocesseur : comme son nom le suggére, c'est en fait une phase « préliminaire » considérée comme ne faisant pas partie du langage lui-même. C'est ce préprocesseur qui est chargé de traiter les lignes du code source dont le premier caractère est #. Il travaille uniquement au niveau du fichier en mode texte, et il ne connaît rien à la syntaxe de C, pas plus qu'à celle des autres langages ! Son rôle est de construire un fichier intermédiaire « préprocessé », c'est-à-dire un fichier où toutes les lignes du fichier initial qui commencent par # ont disparu et ont été remplacées par le résultat de la directive demandée. Les autres lignes du source peuvent avoir été modifiées, notamment celle qui utilisent les chaînes de caractères qui ont été « définies » pour le préprocesseur. Par exemple, avec la macro-définition de NULL (trouvée dans stdio.h) suivante :

```
# define NULL ((void*) 0)
```

la ligne de source suivante :

```
ptr=malloc(10); if (ptr==NULL)
```

sera remplacée par la ligne :

```
ptr=malloc(10); if (ptr==((void*) 0)
```

Attention : c'est bien un remplacement de texte qui est fait ! Pour s'en convaincre, analyser le bout de code [1] suivant :

```
#define UN 1
#define DEUX UN+UN
#define QUATRE DEUX*DEUX
...
if (QUATRE==3) printf("BUG\n");
```

sera remplacée par le préprocesseur par :

```
if (1+1*1+1==3) printf("BUG\n");
```

ce qui, comme la multiplication est prioritaire par rapport à l'addition, donnera une condition vraie ! Il aurait fallu écrire :

```
#define UN 1
#define DEUX (UN+UN)
#define QUATRE (DEUX*DEUX)
```

1. exemple aimablement fourni par Michel Beigbeder.

ce qui aurait donné après traitement :

```
if (((1+1)*(1+1))==3) printf("BUG\n");
```

et donc, une exécution « correcte » !

Revenons à notre ligne #include <stdio.h> ; celle-ci est interprétée par le préprocesseur qui recherche dans des répertoires standard (il existe une autre forme pour la directive include sous la forme #include "stdio.h" : sous cette forme, le préprocesseur recherche d'abord le fichier demandé dans le répertoire courant avant de le rechercher dans les répertoires standard) le fichier dont le nom est indiqué. S'il le trouve, il « inclut » le fichier, en lui faisant subir le même traitement que le fichier initial (traitement des lignes commençant par #, recherche des macros).

On peut générer ce fichier prétraité en utilisant la commande :

```
$ gcc -o toto.i -E toto.c
```

l'option -E du compilateur indique que l'on ne doit effectuer que le prétraitement, l'option -o toto.i indique que le résultat doit être stocké dans le fichier toto.i (sinon, le résultat est affiché sur la sortie standard).

Au total, le fichier initial, une fois traité, devient assez volumineux (7 lignes de code source, plus de 1000 lignes une fois traité). Notons que ce nouveau fichier contient une ligne qui nous intéresse :

```
extern int puts(const char *);
```

C'est une déclaration de prototype de la procédure puts, qui indique le nombre et le type des paramètres de cette procédure, ainsi que le type de son résultat. Le mot-clé extern indique au compilateur que cette procédure est définie « ailleurs » (pas dans le fichier en cours).

A.1.2 Le compilateur, l'assembleur

Une fois le fichier prétraité, c'est maintenant réellement le compilateur C qui va effectuer la phase suivante : il va traduire le fichier source C en un langage intermédiaire, dit langage d'assemblage, qui est spécifique au processeur sur lequel est effectuée la compilation, et qui est encore « lisible » avec un éditeur de texte (on peut écrire directement ses programmes en assembleur !).

C'est durant cette phase que la vérification de l'utilisation à bon escient des procédures est effectuée : toujours dans notre cas, le compilateur peut (grâce au prototype présent dans le fichier prétraité) vérifier que l'utilisation de puts est cohérente : il y a bien un paramètre de type char*.

On peut là encore demander à l'enchaîneur de passes de stopper à cette phase de compilation par la commande :

```
$ gcc -S toto.c
```

l'option -S indiquant de s'arrêter après la compilation. Cette commande construit (sauf en cas d'erreur de syntaxe...) un fichier de nom toto.s, de type texte, contenant la traduction en langage d'assemblage du fichier source.

Enfin, une fois cette compilation effectuée, l'assembleur effectue la traduction en langage machine (non lisible avec un éditeur de texte) : le fichier généré (`toto.o` dans notre exemple, le suffixe `.o` signifiant *object code* ou *code objet*) contient le code machine de la procédure unique déclarée dans notre programme : la procédure `main`. On peut demander à `gcc` de stopper après cette phase d'assemblage par l'option `-c` :

```
$ gcc -c toto.c
```

Ce code objet n'est pas « lisible » avec un éditeur de texte, mais on dispose d'un outil appelé `nm` qui permet d'analyser le contenu d'un code objet. Comme toute commande, elle dispose de nombreuses options, mais on peut se contenter ici de sa fonctionnalité de base :

```
$ nm toto.o
00000000 T main
 U puts
```

Cette commande nous apprend les choses suivantes :
- la commande a reconnu le format du code objet !
- il y a dans ce code un symbole `main` qui est dans la zone `T` comme `Text`, c'est-à-dire la zone de code exécutable (ne pas oublier que la mémoire contient des données et du code !) ; la valeur indiquée `00000000` donne l'adresse mémoire (relative à la zone de texte) où commence ce code exécutable ;
- il y a dans ce code un symbole `puts` qui est dans la zone `U` comme `Undefined`, c'est-à-dire qu'il est fait référence à ce symbole, mais l'assembleur n'a pas trouvé le code correspondant (c'est normal, notre fichier source ne contient pas le code source de `puts`).

À noter que, bien que de très nombreuses procédures soient déclarées dans notre fichier (toutes les procédures définies dans `stdio.h`), seule `puts` est indiquée dans le fichier objet, car c'est la seule réellement utilisée.

A.1.3 L'éditeur de liens

La dernière phase permettant la production d'un programme exécutable est l'*édition de liens*. Cela va consister, dans notre exemple simple :
- à trouver quelque part le code de la procédure `puts` ; cette procédure peut elle-même faire référence à d'autres procédures, qu'il faudra également trouver ;
- à ajouter à notre programme le code de démarrage (ce qui se passe avant la procédure `main`) et le code d'arrêt (ce qui se passe après la procédure `main`) ;

La première opération est faite en consultant un ensemble de codes objets fournis avec l'environnement de développement. Afin d'éviter de consulter de nombreux fichiers, on regroupe ces codes objets en un seul fichier appelé *bibliothèque* (*library* en anglais, et parfois improprement *librairie* en français...). Dans le cas des procédures standard du langage `C`, toutes les procédures sont dans la bibliothèque dite `libc` ou `glibc` dans le cas de `gcc` : la localisation exacte de ce fichier dépend de l'environnement de développement.

La deuxième opération est faite en consultant un autre ensemble de codes objets : sous Linux, ces codes se trouvent dans plusieurs fichiers, nommés `crt1.o`[2], `crti.o` et `crtbegin.o`

2. le préfixe `crt` signifie *C run time*

(avant la procédure main), crtend.o et crtn.o (après la procédure main). Leur localisation exacte dépend, encore une fois, de l'environnement de développement.

A.1.4 Le run-time

On appelle *run-time* le code fourni par le système d'exploitation au moment de l'exécution. En effet, nos programmes vont utiliser les ressources du système (la mémoire, le processeur, les fichiers, les périphériques), et vont donc utiliser les fonctionnalités du système d'exploitation : ces fonctionnalités sont rendues par du code toujours présent en mémoire, qui est le code spécifique du système d'exploitation.

Mais ce n'est pas tout. En effet, si l'on relance la commande nm sur l'exécutable obtenu après édition de liens (toujours le petit exemple toto.c), on a la surprise de voir que de nombreux symboles sont désormais présents dans le fichier, mais que la procédure puts n'est toujours pas définie ! En effet, nm affiche quelque chose ressemblant à :

```
$ nm a.out
..
0804833c T main
         U puts@@GLIBC_2.0
..
```

Ceci s'explique par la motivation suivante : de très nombreux programmes vont utiliser les procédures de la bibliothèque C (tous ceux développés en C, ce qui est le cas de presque tous les utilitaires fournis avec Linux...), et il serait catastrophique aussi bien au niveau de l'encombrement disque (pour stocker les fichiers des programmes exécutables) que de l'encombrement mémoire (lorsque les programmes sont exécutés) de dupliquer ces codes !

Le système Linux, comme la plupart des Unix, a pour cela mis en place un dispositif qui permet d'éviter ces deux inconvénients.

A.1.4.1 Chargement dynamique des bibliothèques

Au moment de l'édition de liens, le code des bibliothèques n'est en fait pas « ajouté » au programme : l'éditeur de liens vérifie simplement que les procédures utilisées dans le programme existent quelque part, c'est-à-dire dans une bibliothèque **accessible au moment de la compilation**, et il mémorise dans le fichier exécutable, pour chaque procédure, la bibliothèque où cette procédure a été trouvée. C'est ce qu'indique l'utilitaire nm : le programme utilise la procédure puts, qui a été trouvée dans la bibliothèque glibc (le 2.0 qui suit indique en plus la version de la bibliothèque).

C'est au tout début de l'exécution du programme que le code des bibliothèques va être recherché et installé en mémoire : on appelle ce mécanisme *chargement dynamique de bibliothèques*, ou encore *dynamic loading of librairies* (DLL). Ce code fait partie de ce qui est exécuté avant la procédure main.

Sur certains systèmes, cette recherche se fait même de façon *paresseuse* (c'est le vrai terme utilisé, traduction de l'anglais *lazy resolution*) : un minimum de travail est fait au démarrage du programme, et c'est à la première utilisation de la procédure que l'on recherche réellement où

elle se trouve (grosso modo, la philosophie est de se dire : pourquoi faire systématiquement un travail alors que l'on peut peut-être s'en passer si la procédure n'est pas utilisée ? d'où le nom de paresseux). Les appels ultérieurs ne posent eux pas de problème.

A.1.4.2 Bibliothèques partagées

Le chargement dynamique des bibliothèques décrit précédemment permet d'éviter de dupliquer le code dans les programmes exécutables (fichiers), ce qui économise de la place disque.

Pour économiser la place mémoire, on utilise un mécanisme complémentaire : les bibliothèques sont de plus partagées entre les différents programmes qui les utilisent. Ceci revient à dire que chaque bibliothèque est au plus présente une fois en mémoire (dès qu'au moins un programme l'utilise).

Pour cette raison, on nomme parfois ces bibliothèques des *objets dynamiques partagés* ou *dynamic shared objects* ou encore *DSO*. On trouve notamment dans le programme exécutable, toujours grâce à la commande `nm` :

```
$ nm a.out
..
08049440 D __dso_handle
..
```

un symbole appelé `__dso_handle`, qui est en fait l'adresse de la tête d'une liste chaînée des bibliothèques partagées dynamiques que le programme utilise.

A.2 Modularité en C

A.2.1 Principe général

Dans le schéma simple, il n'y a qu'un seul fichier source : ce fichier est compilé (prétraité, compilé, assemblé) et l'édition de liens ajoute les procédures de la bibliothèque C, le *C run time*.

Il n'y a aucun problème pour étendre ce processus à plusieurs fichiers sources. Chaque fichier source est, totalement indépendamment des autres, prétraité, compilé, assemblé. Simplement, l'édition de liens traite la collection des fichiers objets générés, et ajoute ensuite la *libc* et le *C run time* !

A.2.2 Un premier exemple

Voici par exemple la construction d'un programme à partir de 2 sources, `toto.c` qui contient :

```
#include <stdio.h>

void toto (int i)
{
  printf("i = %d\n",i);
}
```

et `titi.c` qui contient :

```
extern void toto (int);

int main (int argc,char *argv)
{

  toto(argc);
  toto(3);
}
```

On peut noter :
- la présence dans le premier fichier de l'inclusion de `stdio.h`, nécessaire pour l'utilisation de la procédure `printf` ;
- l'absence dans le deuxième fichier de cette inclusion (aucune procédure de la bibliothèque C n'est utilisée) ;
- la déclaration en `extern` du prototype de la procédure `foo`, utilisée ensuite.

La construction de l'exécutable correspondant peut se faire par la succession des commandes :

```
$ gcc -Wall -c toto.c
$ gcc -Wall -c titi.c
$ gcc toto.o titi.o
```

Bien noter que, pour chaque fichier source, on précise l'option `-c` qui indique de ne pas faire l'édition de liens (chaque fichier ne génère pas à lui tout seul le programme complet). Pour la dernière commande, comme les fichiers ont le suffixe `.o`, `gcc` en déduit que ce sont des fichiers objet et ne fait donc que l'édition de liens.

On aurait également pu construire le programme en une seule commande :

```
$ gcc -Wall toto.c titi.c
```

la différence étant simplement que les codes objet temporaires sont effacés après l'édition de liens.

A.2.3 Utilisation du préprocesseur

On peut également utiliser le préprocesseur : il est ainsi possible de créer ses propres fichiers d'en-tête (les `.h`). Dans le petit exemple précédent, voici une nouvelle version du fichier `titi.c` :

```
#include "toto.h"

int main (int argc,char *argv)
{

  toto(argc);
  toto(3);
}
```

le fichier `toto.h` contenant à son tour :

```
extern int toto (int);
```

Qu'est-ce que l'on a gagné ? Pas grand chose sur cet exemple précis, mais si l'on a de nombreuses procédures définies dans un même fichier, et que ces procédures sont à leur tour utilisées dans plusieurs autres fichiers sources (penser à `stdio.h` qui contient de très nombreuses procédures, utilisées dans de très nombreux sources...), on économise de nombreuses lignes de déclarations remplacées par la seule ligne d'inclusion, et on y gagne en lisibilité !

NB : on a utilisé la seconde forme de la macro d'inclusion où le nom du fichier est entre doubles quotes, on rappelle que dans ce cas, le fichier est d'abord recherché dans le répertoire courant.

A.2.4 Comment découper un programme source ?

Avec le schéma de construction des programmes tel qu'il a été présenté, il n'est pas possible de répartir une procédure C entre deux fichiers sources, et on peut donc avoir deux situations extrêmes :

- le source est intégralement contenu dans un seul fichier quel que soit le nombre de procédures le composant ;
- le source est découpé en autant de fichiers qu'il y a de procédures le composant.

Toute solution intermédiaire entre ces deux extrêmes est techniquement réalisable, et un choix particulier relèvera plutôt des bonnes pratiques de programmation que d'une contrainte liée au langage ou à l'environnement de développement.

A.2.5 Construction automatique des programmes

Lorsque le nombre de fichiers sources pour un même exécutable devient important, il devient assez ardu de construire l'exécutable. Si un fichier est modifié, il est nécessaire de le recompiler, mais il n'est peut-être pas nécessaire de tout recompiler. C'est pour remplir cette tâche particulière que l'environnement Linux propose un outil de construction automatique de programmes appelé `make`.

Cet outil utilise un fichier de configuration appelé `Makefile` ou encore `makefile` : ce fichier contient, avec une syntaxe un peu particulière, les règles de construction. Toujours dans notre exemple simple ci-dessus, voici une façon possible d'écrire ce fichier `Makefile`

```
prog:toto.o titi.o
	gcc -o prog toto.o titi.o
toto.o:toto.c
	gcc -Wall -c toto.c
titi.o:titi.c toto.h
	gcc -Wall -c titi.c
```

Chacune des lignes 1, 3 et 5 définit une règle de dépendance : cette règle sera appliquée si le programme `make` se rend compte que l'un des fichiers dont le nom est situé après le caractère `:` a été modifié postérieurement à celui dont le nom est indiqué avant ce caractère. Ainsi, `prog` dépend de `toto.o` et de `titi.o`, `toto.o` dépendant à son tour de `toto.c` et `titi.o` dépendant de `titi.c` et de `toto.h`.

Le fait d'avoir ajouté `toto.h` dans la liste des dépendances de `titi.o` provient de l'inclusion de `toto.h` dans `titi.c` : si l'on modifie `toto.h`, il faut bien recompiler `titi.c` !

Noter enfin que l'espace avant le mot `gcc` sur chacune des trois lignes 2, 4 et 6 **doit être** une tabulation, et non une succession d'espaces.

Une fois ce fichier écrit, il suffit de taper la commande `make prog` et le programme `make` reconstruira si besoin le fichier `prog` :

```
$ make prog
gcc -Wall -c toto.c
gcc -Wall -c titi.c
gcc -o prog toto.o titi.o
$ make prog
make: `prog' is up to date.
$ gedit toto.c
$ make prog
gcc -Wall -c toto.c
gcc -o prog toto.o titi.o
```

Annexe B

La saisie de données au clavier

Avant-propos

Le but de ce petit document est d'indiquer quelques fonctionnalités utilisables pour le traitement des saisies d'information au clavier. Ceci concerne l'environnement Unix de façon générale, et Linux en particulier. Le rédacteur de cette note avoue sa méconnaissance du fonctionnement précis sous Windows...

B.1 Rappel

Chaque programme `C` dispose d'un flot dit « entrée standard » connu sous le sobriquet de `stdin` (macro-définition présente dans le fichier d'en-tête `stdio.h`). C'est sur ce flot accessible en lecture seulement que la procédure `scanf` (ainsi que la macro/procédure `getchar`) lit ses données.

Lorsque le programme `C`, une fois compilé, est lancé depuis un émulateur de terminal, cette entrée standard est « connectée » au clavier (de la même façon que la sortie standard est « connectée » vers l'écran) : lors d'une lecture sur `stdin`, l'utilisateur tape sur les touches du clavier, et lorsqu'il appuie sur la touche `Return` ou `Entrée`, ce qu'il a frappé est disponible pour le programme.

B.2 Fonctionnement par défaut du terminal

Vous avez sûrement constaté que les données transmises vers le programme n'étaient pas exactement la succession des caractères tapés au clavier :

- l'utilisation de la touche `Backspace` ou `Ctrl-H` (cette notation signifie : appuyer d'abord sur la touche `Ctrl` et tout en la maintenant enfoncée, appuyer sur la touche `H`) a pour effet d'effacer le dernier caractère saisi ;
- la touche `Delete` ou `Suppr` efface totalement la ligne ;
- l'utilisation de la combinaison `Ctrl-C` interrompt le déroulement du programme.

Vous avez aussi constaté que lorsque vous tapez une touche « normale », vous voyez apparaître le caractère frappé à l'écran.

Tout ceci est l'œuvre d'un composant du système d'exploitation que l'on nomme *driver tty* [1] ou *pilote de terminal*. Ce pilote de terminal a un rôle triple (au moins) :

- c'est lui qui est chargé d'afficher les caractères tapés (fonction d'*écho*) ;
- c'est lui qui est chargé de l'édition de la ligne en cours de saisie (possibilité d'effacer un caractère, un mot, la ligne complète) ; cette fonctionnalité particulière est souvent désignée dans la documentation sous le vocable de *traitement canonique de l'entrée* ou *input canonical processing* ;
- c'est lui qui est chargé de traiter les caractères de contrôle (le `Ctrl-C` qui interrompt le programme, mais il en existe d'autres). On parle ici de *traitement des signaux* ou de *signal processing*.

B.2.1 Traitement canonique

Le traitement canonique du pilote de terminal fonctionne *en mode ligne* : les caractères tapés au clavier sont utilisés pour construire une ligne complète de données, et c'est lors de la frappe de la touche `Return` que la ligne ainsi constituée devient disponible pour le programme. Rappelons que le caractère « fin de ligne » est présent dans les données envoyées au programme.

Si l'on veut forcer l'envoi vers le programme d'une ligne incomplète (non terminée par une fin de ligne), on peut le faire en tapant la séquence `Ctrl-D` (ce caractère n'est pas ajouté à la ligne). **Attention :** si le pilote clavier reçoit une ligne vide, c'est-à-dire en tapant `Ctrl-D` en début de ligne, il génèrera un indicateur de fin de fichier pour le programme qui tente la lecture. Dans ce cas, `scanf` (ou `getchar`) retourne la valeur `EOF` (définie dans le fichier d'en-tête `stdio.h` comme étant la valeur numérique -1). Certains interpréteurs de commandes considèrent cette fin de fichier comme une volonté de l'utilisateur de terminer le traitement, et terminent alors leur fonctionnement !

Notons enfin que c'est toute la ligne qui est envoyée en une seule fois au programme : ainsi, si l'on effectue une boucle de lecture par `getchar`, le programme, lors du premier appel, va attendre que la ligne ait été saisie et terminée par la touche `Entrée` ; tous les appels successifs à `getchar` seront immédiatement satisfaits jusqu'à ce que le caractère fin de ligne ait été lu (pas d'attente) ; le `getchar` suivant provoquera une nouvelle attente, etc.

Les diverses fonctionnalités du traitement canonique sont accessibles par certains caractères tapés au clavier, dits « caractères spéciaux ». Ces caractères spéciaux sont indiqués ci-dessous, avec leur nom symbolique :

erase caractère spécial effaçant le dernier caractère tapé ; en général, `Ctrl-H` ou `Backspace` ;

kill caractère spécial effaçant la totalité de la ligne ; en général, `Ctrl-U` ;

eof (end of file) caractère spécial forçant l'envoi de la ligne en cours d'édition ; en général, `Ctrl-D` ;

---

1. le nom `tty` est le diminutif de *teletype* désignant une imprimante connectée à l'ordinateur par une ligne série ; ce nom est resté pour désigner le pilote de la ligne série, sur lesquelles on a connecté ensuite des terminaux alphanumériques ; ces terminaux sont maintenant remplacés par des émulateurs de terminal en environnement multi-fenêtré, mais le nom initial est resté !

eol (end of line) caractère spécial marquant la fin de ligne ; en général, `Ctrl-J` ou `Return` ou `Entrée` ;

eol2 autre caractère spécial marquant la fin de ligne ; en général, non défini ;

werase (word erase) caractère spécial effaçant le dernier « mot » (efface les caractères précédemment saisi jusqu'au premier blanc) ; en général, `Ctrl-W` ;

rprnt (reprint) caractère spécial permettant de réafficher la totalité des caractères déjà saisis dans la ligne d'édition ; en général, `Ctrl-R` ;

lnext caractère spécial permettant de « banaliser » le caractère suivant (utilise pour saisir les caractères spéciaux dans la ligne !) ; en général, `Ctrl-V` ; par exemple, si l'on souhaite ajouter le caractère `Ctrl-H` dans la ligne, on tape d'abord `Ctrl-V` puis `Ctrl-H`, si l'on souhaite ajouter le caractère `Ctrl-V`, on tape deux fois `Ctrl-V` ;

B.2.2 Traitement des signaux

Le pilote de terminal offre la possibilité à l'utilisateur de générer des signaux pour les programmes en cours d'exécution. Ce sont également des caractères spéciaux qui déclenchent ces signaux. On dispose ainsi :

- d'un signal d'interruption dit `SIGINT` ; en général, c'est la combinaison `Ctrl-C` qui déclenche l'envoi de ce signal ; la plupart des programmes, lorsqu'ils recoivent ce signal, terminent prématurément et immédiatement leur fonctionnement ;
- d'un signal dit `SIGQUIT` ; en général, c'est la combinaison `Ctrl-\` qui déclenche l'envoi de ce signal ; la plupart des programmes, lorsqu'ils recoivent ce signal, terminent prématurément et immédiatement leur fonctionnement (comme pour `SIGINT`) et génèrent une copie de leur espace mémoire dans un fichier *core* ;
- d'un signal de suspension dit `SIGTSTP` ; en général, c'est la combinaison `Ctrl-Z` qui déclenche l'envoi de ce signal ; ceci permet de suspendre l'exécution d'un programme, et redonne la main à l'interpréteur de commandes ; on peut poursuivre ultérieurement l'exécution du programme stoppé (commande `bg` et `fg` de l'interpréteur) ;

On peut insérer ces caractères spéciaux dans la ligne de saisie en les précédant du caractère d'échappement (`lnext`) décrit dans le paragraphe précédent : par exemple, la séquence `Ctrl-V Ctrl-C` permet d'ajouter le caractère `Ctrl-C` (code ASCII 3) dans la ligne de saisie.

B.2.3 Traitement de l'écho

Par défaut, tout caractère tapé au clavier est affiché (en cours de saisie) sur l'écran. **Attention :** cet écho ne doit pas être confondu avec l'affichage des caractères écrits par le programme sur la sortie standard, également connectée à l'écran !

D'autres fonctions sont offertes par cette fonctionnalité d'écho :

echoe lors d'une frappe du caractère d'effacement, l'écho génère la séquence : espace arrière, espace, espace arrière (permet d'effacer « physiquement » le caractère du terminal) ;

echok effectue un écho d'un saut de ligne après la frappe d'un caractère d'effacement de la ligne ;

echoctl effectue un écho des caractères de contrôle (les combinaisons `Ctrl-X`) sous la forme `^X`.

B.3 Quelques incidences sur la lecture de données

B.3.1 Un petit problème

Il faut bien avoir à l'esprit le mode de fonctionnement par défaut du pilote de terminal lorsque l'on souhaite demander interactivement des informations à l'utilisateur. Prenons l'exemple de code suivant :

```
int n;
char c;
...
printf("Donner la valeur de n ?");
scanf("%d",&n);
printf("Taper un caractere pour continuer :");
scanf("%c",&c);
...
```

À l'exécution, le programme attend que l'utilisateur donne la valeur de `n`, puis affiche le message `Taper..` et continue son exécution sans attendre !

Explication : en fait, c'est le traitement par ligne qui est responsable de ce petit problème. Lors de la saisie de la valeur numérique `n`, l'utilisateur tape au clavier les chiffres décimaux qui composent cette valeur, et valide sa ligne en tapant sur la touche `Entrée` : la ligne, y compris le caractère de fin de ligne, est donc transmise au programme.

Lorsque le premier `scanf` analyse les données saisies, il interprète correctement les chiffres décimaux pour déterminer la valeur de `n` : cette analyse se stoppe sur le caractère fin de ligne, **qui est remis dans les données en entrée**. Lors du second `scanf`, comme il reste des données de la lecture précédente, on commence par analyser celles-ci : on n'a besoin que d'un seul caractère (format `%c`) et c'est donc le caractère fin de ligne qui est immédiatement lu, d'où la poursuite sans attente de l'exécution du programme !

B.3.2 Comment éviter cela ?

B.3.2.1 La procédure `fflush`

La procédure `fflush` permet de forcer le vidage d'un flot : par exemple, lorsque l'on écrit des données sur le flot `stdout`, on ne voit rien apparaître à l'écran tant que l'on n'écrit pas une fin de ligne (ou que l'on n'effectue pas une lecture au clavier). L'utilisation de l'instruction `fflush(stdout)` permet de forcer ce vidage.

Sur certains systèmes, on peut également utiliser `fflush` sur un flot ouvert en lecture : l'effet est alors de « vider » les données transmises au programme mais non encore lues. C'est

précisément ce que l'on cherche à faire. Hélas : ce fonctionnement de `fflush` ne fait pas partie de la norme ANSI-C, qui précise simplement que sur les flots ouverts en lecture, l'effet de `fflush` est indéterminé. Linux s'en tient à la stricte implémentation ANSI, et un appel à `fflush` sur un flot ouvert en écriture retourne un code d'erreur.

On dispose sous Linux d'une procédure équivalente, nommée `__fpurge` : bien qu'utilisable, le manuel précise qu'elle n'est « ni standard ni portable ». À utiliser avec modération, donc. . .

B.3.2.2 Lecture en deux temps

On peut pallier ces inconvénients par la méthode dite de lecture en deux temps :

- toute lecture au clavier saisit une ligne complète dans un tableau de caractères ;
- l'analyse se fait ensuite sur le tableau lu ;

La première phase peut s'effectuer par la procédure `fgets` (l'utilisation de `gets` est totalement à éviter car source de nombreuses failles de sécurité). Pour le seconde, on peut avantageusement remplacer tout usage de `scanf` par un appel à `sscanf`, procédure de comportement identique à `fscanf` sauf qu'au lieu d'indiquer en premier paramètre un flot, on indique un tableau de caractères.

Le petit exemple ci-dessus devient, avec cette méthode :

```
#define LINEMAX 1024
int n;
char c,line[LINEMAX];
...
printf("Donner la valeur de n ?");
fgets(line,LINEMAX,stdin);
sscanf("%d",&n);
printf("Taper un caractere pour continuer :");
fgets(line,LINEMAX,stdin);
sscanf(line,"%c",&c);
```

Lors du deuxième appel à `fgets`, les données présentes dans le tableau `line` et non encore lues sont tout simplement écrasées !

B.4 Programmation du pilote du terminal

Le pilote de terminal, dont le nombre de fonctionnalités est très important (la page de manuel de ce pilote `termios(3)` est souvent la plus longue du manuel Unix !), est totalement configurable. Ce qui a été présenté ci-dessus est le fonctionnement par défaut, et on peut donc à loisir le modifier.

B.4.1 Traitement non-canonique

On peut ainsi notamment supprimer le traitement canonique. Dans ce cas, le mode de fonctionnement par ligne n'est plus actif, et les caractères spéciaux associés (effacement d'un ca-

ractère, de la ligne) sont interprétés comme des caractères standard. Attention cependant : les caractères permettant de générer les signaux sont conservés !

En mode non-canonique, les caractères saisis au clavier sont transmis au programme selon le mode suivant :

- le pilote satisfait (transmet les données) les demandes de lecture dès qu'un nombre minimal de caractères a été saisi ; attention : le nombre de caractères transmis au programme peut être inférieur au nombre de caractères demandés ;
- le pilote dispose d'un *timer* ou intervalle de temps : si cet intervalle de temps est dépassé après la lecture d'au moins un caractère, le pilote n'attend pas que le nombre minimal de caractères soit reçu pour transmettre les données au programme (ceci permet d'éviter les attentes trop longues).

Ces deux paramètres sont réglables : le timer a une précision du dixième de seconde et peut aller jusquà 25.5 secondes. Pour le pilote, ils sont considérés comme des pseudo-caractères spéciaux, dénommés `min` et `time`.

B.4.1.1 Exemple important

Il est fréquent de vouloir lire les caractères 1 par 1, dès qu'ils sont tapés au clavier, et sans attente de la touche `Entrée` : il suffit pour cela de placer le pilote de terminal en mode non-canonique, en réglant le nombre minimal de caractères à 1 et le timer à 0.

B.4.2 Écho, signaux

On peut également invalider la fonctionnalité d'écho : ceci invalide presque toutes les fonctionnalités associées, mais on conserve la possibilité de faire un écho du seul caractère `Entrée`. Voir la documentation pour cela.

De la même façon, on peut invalider la fonctionnalité de traitement des signaux.

B.5 Dans la pratique

La configuration du pilote de terminal peut se faire de deux façons :

- l'utilisation de la commande `stty(1)`, lancée à partir de l'interpréteur de commandes, modifie les paramètres du terminal associé à l'interpréteur ; ces paramètres modifiés le sont également pour tout programme lancé par ce dernier ;
- l'utilisation des procédures de bibliothèque `termios(3)`, directement à partir du programme.

Il est à noter que certains interpréteurs (`tcsh` notamment), sauvegardent les paramètres du terminal avant toute commande, et les rétablissent ensuite : la première méthode (par la commande `stty`) est donc totalement inopérante, et c'est donc vers la deuxième qu'il faut se tourner.

B.5.1 Quelques utilisations de `stty`

Attention : la syntaxe indiquée ici est celle disponible sous Linux. Sous d'autres environnements Unix (Solaris, IRIX, HP-UX), il peut y avoir quelques différences. Toujours se reporte au manuel en ligne.

La suppression du traitement canonique se fait par la commande :

```
$ stty -icanon
```

ou la commande équivalente

```
$ stty cbreak
```

Pour revenir en mode canonique, on utilise l'une des deux versions :

```
$ stty icanon
$ stty -cbreak
```

Si l'on souhaite, en mode non-canonique, préciser les paramètres :

```
$ stty -icanon min 1 time 0
```

La suppression (et le rétablissement) du traitement des signaux se font par :

```
$ stty -isig
$ stty isig
```

La suppression (et le rétablissement) de l'écho se font par :

```
$ stty -echo
$ stty echo
```

Enfin, on dispose des modes combinés `raw` (brut) et `cooked` (cuisiné). Si l'on a mis son terminal dans un état lamentable, on peut restaurer une configuration valable par la commande

```
$ stty sane
```

B.5.2 Configuration en C du pilote de terminal

On a besoin d'une structure de données spécifique, nommée `struct termios` pour effectuer cette configuration. Notre but n'est pas d'expliquer en détail le contenu de cette structure, ni comment on l'utilise. Les personnes intéressées peuvent consulter le manuel en ligne, ou consulter les enseignants.

On trouvera néanmoins ci-dessous deux procédures : l'une sauvegarde la configuration du terminal, et le reconfigure de façon à passer en mode non-canonique, sans écho et sans gestion des signaux. Le nombre minimal de caractères est positionné à 1, le timer à 0. La seconde procédure permet de restaurer la configuration initiale (utile pour ne pas laisser le terminal dans un état peu utilisable…). Ces procédures sont utilisées dans un petit programme qui affiche tous les caractères tapés : c'est la touche `Delete` ou `Suppr` qui permet de terminer le fonctionnement. Vous pouvez le compiler, le lancer, et essayer de taper `Ctrl-C`, `Backspace`, `Entrée`…

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

/* cette procedure reconfigure le terminal, et stocke */
/* la configuration initiale a l'adresse prev */
int reconfigure_terminal (struct termios *prev)
{
struct termios new;
  if (tcgetattr(fileno(stdin),prev)==-1) {
    perror("tcgetattr");
    return -1;
  }
  new.c_iflag=prev->c_iflag;
  new.c_oflag=prev->c_oflag;
  new.c_cflag=prev->c_cflag;
  new.c_lflag=0;
  new.c_cc[VMIN]=1;
  new.c_cc[VTIME]=0;
  if (tcsetattr(fileno(stdin),TCSANOW,&new)==-1) {
    perror("tcsetattr");
    return -1;
  }
  return 0;
}

/* cette procedure restaure le terminal avec la */
/* configuration stockee a l'adresse prev */
int restaure_terminal (struct termios *prev)
{
  return tcsetattr(fileno(stdin),TCSANOW,prev);
}

/* exemple d'utilisation */
int main (int argc,char *argv[])
{
struct termios prev;
int nb,c;

  if (reconfigure_terminal(&prev)==-1)
    return 1;
  for (nb=0;;) {
    c=getchar();
```

```
      nb++;
      (void) printf("carac[%d]=(%d,%o,%x)",nb,c,c,c);
      if (c==127) {
        printf(" char=DEL\n%d caracteres\n",nb);
        break;
      }
      if (c>=32)
        printf(" char=%c\n",c);
      else
printf("\n");
    }
    if (restaure_terminal(&prev)==-1)
      return 1;
    return 0;
  }
```

B.6 Modification de l'entrée standard

Ainsi que cela a été indiqué, tout programme lancé à partir d'un interpréteur de commandes se retrouve avec son entrée standard connectée sur le terminal de l'interpréteur. Il est toutefois possible de modifier ce fonctionnement.

B.6.1 Redirection à partir d'un fichier

On peut utiliser en lieu et place du clavier un fichier dans lequel on a écrit toutes les réponses/données nécessaires pour le programme. Par exemple, imaginons un programme qui demande à l'utilisateur un nombre de données, puis saisit le nombre indiqué de valeurs numériques. On peut alors créer un fichier `prog.in` (le nom n'a aucune importance !) qui contient :

```
3
123
345
206
```

et lancer le programme en indiquant que la saisie des paramètres se fait dans le fichier `prog.in`, en lançant le programme par :

```
$ ./prog < prog.in
```

NB : si ce sont les résultats affichés par le programme sur la sortie standard que l'on souhaite conserver dans un fichier `prog.out` (encore une fois, le nom n'a aucune importance) au lieu de les écrire sur le terminal, on utilise la syntaxe suivante :

```
$ ./prog > prog.out
```

les deux opérations étant combinables :

```
$ ./prog < prog.in > prog.out
$ ./prog > prog.out < prog.in
```

B.6.2 Redirection à partir des résultats d'un programme

Il peut y avoir des cas où le fonctionnement suivant est attendu : un premier programme affiche des résultats (sur sa sortie standard), ces résultats devant être saisi **sous la même forme** par un second programme pour obtenir le résultat final. On peut penser à stocker les résultats intermédiaires dans un fichier, sur lequel on redirige l'entrée standard du second programme, comme dans l'exemple suivant :

```
$ ./prog1 > resultats
$ ./prog2 < resultats
$ rm resultats
```

On peut en fait éviter de créer ce fichier intermédiaire, et tout effectuer en une seule commande :

```
$ ./prog1 | ./prog2
```

Attention à bien respecter l'ordre ! On peut même généraliser ceci à n programmes :

```
$ ./prog1 | ./prog2 | ./prog3 | ./prog4
```

où pour tout `i`, la sortie standard de `progi` est connectée sur l'entrée standard de `progi+1`. En terminologie Unix, une telle succession de programmes porte le nom de *tube* (*pipe* en anglais), les programmes intermédiaires (ni le premier, ni le dernier) prenant le nom de *filtres*. Il existe de très nombreux filtres, qui, en les combinant astucieusement, permettent d'obtenir une grande variété de résultats.

Index

acyclique, 81
adjacent, 80
adresse, 60
allocation dynamique (mémoire), 46
append, 10
arborescence, 85
arbre, 85
arbre de recouvrement, 83
arbre recouvrant, 83
arc, 79, 80
arête, 79, 80

big-endian (gros-boutiste), 31
binaire (fichier), 28
buffer, 32

calloc, 48
chaîne, 80
champ (de structure), 40
chemin, 80
circuit, 80
clearerr, 22
codes d'erreur, 12
complexité, 65
connexe, 81
connexité, 81
copie de liste, 72
cycle, 81

définition de type (typedef), 51
degré, 81
dépiler, 76
diamètre, 81
distance, 80

empiler, 76
End Of File, 10
enregistrement, 39
entrée standard, 9
entrées-sorties, 7
enum (énumération), 52
énumération (enum), 52
EOF, 10
errno, 21
exit, 11
exponentiel (algorithme), 66

fclose, 10
feof, 22
fermeture transitive, 83, 93
ferror, 22
fgetc, 16
fgets, 16
fichier, 8
fichier (binaire), 28
fichier (régulier), 28
fichier (texte), 28
fichiers, 7
FIFO, 77
FILE, 9
file, 76
fin de liste, 63
flot, 9, 12
fopen, 10, 11
format (d'écriture), 14
format (de lecture), 18
fprintf, 13
fputc, 13
fputs, 13
fread, 30
free, 50
freopen, 11
fscanf, 16
fseek, 33
ftell, 33

fwrite, 29

getc, 16
getchar, 16
gets, 16
grand O, 65
graphe, 79
graphe clairsemé, 80
graphe dense, 80
graphe orienté, 80
graphe partiel, 83
gros-boutiste (big-endian), 31

head, 20

LIFO, 76
linéaire (algorithme), 66
liste, 60
liste (copie), 72
liste (parcours), 61
liste (réversion), 73
liste chaînée, 60
liste d'adjacence, 81
little-endian (petit-boutiste), 31
logarithmique (algorithme), 66

malloc, 46
man, 9
Markov, 85
matrice d'adjacence, 81
mémoire (allocation dynamique), 46

NULL, 9, 63

O (notation), 65
ordre, 80

parcours (de graphe), 89
parcours (de liste), 61
perror, 21
petit-boutiste (little-endian), 31
pile, 76
point, 79
pointeur, 60
pointeur (sur structure), 44
prédécesseur, 80
printf, 13
producteur-consommateur, 12
profondeur, 89
puits, 87, 93
putc, 13
putchar, 13
puts, 13

quadratique (algorithme), 66
queue de liste, 63

racine, 85
record, 39
régulier (fichier), 28
remove, 22
rename, 22
réversion de liste, 73
rewind, 33
Roy-Warshall (algorithme), 93

scanf, 16, 18
sink, 87, 93
sizeof, 47
solitaire, 79
sommet, 79
sortie standard, 9
source, 87, 93
sparse, 80
sprintf, 13
sscanf, 16
stderr, 9
stdin, 9
stdio.h, 9
stdout, 9
struct, 40
structure de données, 39
successeur, 80

tampon mémoire, 32
tête de liste, 60
texte (fichier), 28
tmpfile, 22
tri par fusion, 75
typedef (définition de type), 51

ungetc, 16

union, 53

valué, 81

Bibliographie

[1] Jean-Jacques Girardot and Marc Roelens. Introduction à l'informatique. `http://kiwi.emse.fr/INTROINFO/`, Septembre 2010.

[2] Jean-Jacques Girardot and Marc Roelens. Structures de données et Algorithmes en C. `http://kiwi.emse.fr/POLE/SDA/`, Octobre 2010.

[3] Donald E. Knuth. Big omicron and big omega and big theta. *SIGACT News*, April-June 1976.

Table des matières

I Cours 3

Introduction 5

1 Entrées-sorties en C 7

1.1 Cours . . . 7
1.1.1 Introduction . . . 7
1.1.2 Représentation des données dans un fichier . . . 7
1.1.3 Bibliothèque standard des entrées-sorties . . . 8
1.1.4 Définition des flots . . . 9
1.1.5 Flots et fichiers . . . 9
1.1.5.1 Création d'un flot à partir d'un fichier . . . 9
1.1.5.2 Destruction d'un flot . . . 10
1.1.5.3 Réouverture d'un flot . . . 11
1.1.6 Quelques propriétés des flots . . . 12
1.1.6.1 Unité d'échange avec un flot . . . 12
1.1.6.2 Accès séquentiel . . . 12
1.1.7 Opérations d'écriture dans un flot . . . 12
1.1.7.1 Écriture d'octets . . . 13
1.1.7.2 Écriture de chaînes . . . 14
1.1.7.3 Écriture formatée . . . 14
1.1.8 Opérations de lecture dans un flot . . . 16
1.1.8.1 Lecture en mode octet . . . 17
1.1.8.2 Lectures de chaînes . . . 17
1.1.8.3 Lecture formatée . . . 18
1.1.9 Un exemple : « head » . . . 20
1.1.10 Autres procédures . . . 21
1.1.10.1 Gestion des erreurs . . . 21
1.1.10.2 Autres procédures de manipulation de flots . . . 22
1.1.10.3 Autres procédures de manipulations de fichiers . . . 22
1.1.11 Un exemple : « high scores » . . . 22
1.1.12 Un exemple, « head, 2 » . . . 23
1.2 Compléments de cours . . . 24
1.2.1 Caractères et chaînes de caractères en C . . . 24
1.2.1.1 Le type `char` . . . 24

1.2.1.2 Les constantes caractères . . . 25
1.2.2 Chaînes de caractères . . . 26
1.2.2.1 Le type chaîne de caractères . . . 26
1.2.2.2 Constantes chaînes de caractères . . . 26
1.2.2.3 Particularités . . . 27
1.2.3 Fichiers texte et fichiers binaires . . . 28
1.2.3.1 Écriture en format binaire . . . 29
1.2.3.2 Lecture en format binaire . . . 30
1.2.4 Gros-boutistes et petits-boutistes . . . 31
1.2.5 Tampons en mémoire . . . 32
1.2.6 Positionnement dans un flot . . . 32
1.3 Travaux pratiques . . . 33
1.3.1 Arguments de la procédure `main` . . . 33
1.3.2 Consulter le manuel . . . 33
1.3.3 Conversion de nombres . . . 34
1.3.4 Écriture d'entiers . . . 34
1.3.5 Nombre de caractères et de lignes dans un fichier . . . 35
1.4 Exercices à faire… s'il reste du temps . . . 35
1.4.1 Conversion hexadécimale . . . 35
1.4.2 Fichiers : format texte ou binaire . . . 36
1.4.3 Format de fichier et manipulation de matrices . . . 36

2 Structures de données en C **39**
2.1 Cours . . . 39
2.1.1 Introduction . . . 39
2.1.2 Types structurés en C . . . 39
2.1.2.1 Définition d'un type structuré . . . 39
2.1.2.2 Structures et tableaux, structures imbriquées . . . 40
2.1.2.3 Accès aux champs d'une structure . . . 42
2.1.3 Structures et procédures . . . 43
2.1.4 Structures référençant des structures . . . 43
2.1.5 Allocation dynamique de structures et tableaux . . . 45
2.1.5.1 La procédure malloc . . . 46
2.1.5.2 La procédure calloc . . . 48
2.1.5.3 La procédure free . . . 50
2.1.6 Définition de type nommé . . . 51
2.2 Compléments de cours . . . 52
2.2.1 Unions et types énumérés . . . 52
2.2.1.1 Énumérations . . . 52
2.2.1.2 Unions . . . 53
2.3 Travaux pratiques . . . 55
2.3.1 Manipulation de fichiers de voitures . . . 55
2.3.1.1 Liminaire . . . 55
2.3.1.2 Procédure de lecture du fichier . . . 56

2.3.1.3 Procédure d'écriture d'un fichier de voitures 56
2.3.1.4 Conclusion 56
2.3.2 Gestion de comptes bancaires 56
2.3.2.1 Liminaire 57
2.3.2.2 Analyse et conception 57
2.3.2.3 Implémentation 57
2.3.2.4 Pour conclure, s'il reste du temps... 57

3 Listes 59
3.1 Cours 59
3.1.1 Introduction 59
3.1.2 Listes chaînées 59
3.1.2.1 Présentation 59
3.1.2.2 Procédures de manipulation de listes 60
3.1.2.3 Longueur d'une liste 61
3.1.2.4 Impression du contenu d'une liste 61
3.1.2.5 Création d'élément 62
3.1.3 Complexité 65
3.1.4 Évaluation des complexités 66
3.1.5 Complexité des opérations usuelles 67
3.2 Travaux pratiques 68
3.2.1 Création de liste 68
3.2.2 Recherche de nombre 68
3.2.3 Suppression d'un élément 68
3.2.4 Création d'une liste triée 69
3.2.5 Gestion d'ensemble 69

4 Listes, files et piles 71
4.1 Cours 71
4.1.1 Algorithmique des listes 71
4.1.1.1 Copie de liste 71
4.1.1.2 Réversion d'une liste 73
4.1.1.3 Tri de listes 73
4.1.2 Pile 76
4.1.3 File 76
4.2 Travaux pratiques 77
4.2.1 Tri de liste 77
4.2.2 Réalisation de files 77

5 Graphes 79
5.1 Cours 79
5.1.1 Introduction 79
5.1.2 Concepts associés 80
5.1.3 Représentation des graphes 81

5.1.3.1 Matrice d'adjacence 81
5.1.3.2 Liste d'adjacence 81
5.1.3.3 Comparaison des représentations 82
5.1.4 Problèmes sur graphes 82
5.1.4.1 Parcours de graphe 83
5.1.4.2 Fermeture transitive 83
5.1.4.3 Arbre de recouvrement 83
5.1.4.4 Circuits 83
5.1.5 Arbres 85
5.2 Travaux pratiques 86
5.2.1 Représentation de graphes 86
5.2.2 Recherche de circuit 87
5.2.3 Calcul de connexité 87

6 Algorithmes sur graphes 89
6.1 Cours 89
6.1.1 Parcours de graphe 89
6.1.1.1 Parcours en profondeur 89
6.1.1.2 Choix des structures de données 90
6.1.1.3 Algorithme général de parcours 90
6.1.1.4 Exemple : recherche de chemin 91
6.1.1.5 Exemple : sommets accessibles depuis un sommet donné . . 92
6.2 Travaux pratiques 93
6.2.1 Fermeture transitive 93
6.2.2 Recherche de circuit 93

II Annexes 95

A Modularité des programmes en C 97
A.1 Construction d'un programme C 97
A.1.1 Le préprocesseur 98
A.1.2 Le compilateur, l'assembleur 99
A.1.3 L'éditeur de liens 100
A.1.4 Le run-time 101
A.1.4.1 Chargement dynamique des bibliothèques 101
A.1.4.2 Bibliothèques partagées 102
A.2 Modularité en C 102
A.2.1 Principe général 102
A.2.2 Un premier exemple 102
A.2.3 Utilisation du préprocesseur 103
A.2.4 Comment découper un programme source ? 104
A.2.5 Construction automatique des programmes 104

B La saisie de données au clavier **107**
B.1 Rappel 107
B.2 Fonctionnement par défaut du terminal 107
B.2.1 Traitement canonique 108
B.2.2 Traitement des signaux 109
B.2.3 Traitement de l'écho 109
B.3 Quelques incidences sur la lecture de données 110
B.3.1 Un petit problème 110
B.3.2 Comment éviter cela ? 110
B.3.2.1 La procédure `fflush` 110
B.3.2.2 Lecture en deux temps 111
B.4 Programmation du pilote du terminal 111
B.4.1 Traitement non-canonique 111
B.4.1.1 Exemple important 112
B.4.2 Écho, signaux 112
B.5 Dans la pratique 112
B.5.1 Quelques utilisations de `stty` 113
B.5.2 Configuration en C du pilote de terminal 113
B.6 Modification de l'entrée standard 115
B.6.1 Redirection à partir d'un fichier 115
B.6.2 Redirection à partir des résultats d'un programme 116

Index **117**

Bibliographie **121**

Table des matières **123**