École Nationale Supérieure
de Techniques Avancées

Programmation en C

Pierre-Alain Fouque et David Pointcheval

E-mail : Pierre-Alain.Fouque@ens.fr
Web : http://www.di.ens.fr/~fouque/

Table des matières

Liste des Programmes

Table des figures

Avant-Propos

Ce document regroupe l'essentiel pour gérer un (gros) projet en C sous un environnement de type UNIX. Mais avant tout, à quoi sert un langage de programmation, tel que le langage C, à l'heure où les ordinateurs obéissent à la voix de leur maître ?

Un ordinateur présente une capacité de calcul énorme. En effet, les machines actuelles atteignent ou dépassent allègrement le milliard d'instructions à la seconde, permettant ainsi d'effectuer des tâches répétitives ou *a priori* longues et fastidieuses, aisément et rapidement.

Cependant, le langage naturel de l'ordinateur (dit *langage machine*) est difficilement compréhensible par l'homme. D'un autre côté, le langage naturel de l'homme est souvent ambigu, plein de sous-entendus, et donc difficile à interpréter par une machine. Le langage C, comme tout langage de programmation, est un moyen de communication avec l'ordinateur, plus abordable. En effet, un interprète, le *compilateur*, va se charger de traduire votre programme C (dit *fichier source*) en langage machine (votre *exécutable*).

Il ne faut donc pas perdre de vue que votre programme servira à commander ...un ordinateur, donc sans imagination ni bonne volonté. Les ordres devront alors être clairs, nets et précis, et surtout sans ambiguïté.

Ainsi, certaines règles doivent-elles être respectées. Ce document tente de recenser les règles essentielles, mais aussi les outils mis à votre disposition pour vous aider à mener à bien un projet important.

Remarque : ce cours ne se prétend pas être un document exhaustif en la matière, mais souhaite apporter, à un programmeur débutant, les bases de la programmation en C, et plus spécifiquement avec `gcc` sous Linux. Ainsi, la fonction `man` est d'une aide précieuse. En effet, la page 3 du " man " contient la description de toutes les fonctions C :
> `man 3 printf`
donne le prototype de toutes les fonctions liées au formattage de sortie.

De plus, ce document ne suit pas le cours magistral, mais tente plutôt de regrouper sous un même chapitre les objets de même type, en se concentrant sur l'essentiel. Aussi, certaines omissions sont volontaires pour éviter les confusions, ainsi que l'utilisation d'astuces très spécifiques au langage C. En effet, le langage C permet beaucoup de liberté, en comparaison au Pascal par exemple. Ce document est un peu plus strict, afin de permettre un passage plus aisé, ultérieurement, à un autre langage (tel le langage JAVA).

Pour toute information :
E-mail : `Pierre-Alain.Fouque@ens.fr`
Web : `http://www.di.ens.fr/~fouque/`
Cours : `http://www.di.ens.fr/~fouque/enseignement/ensta/`

Chapitre 1

Introduction

Sommaire

Comme nous venons de le voir, un langage de programmation sert de moyen de communication avec un ordinateur, qui ne comprend que le langage machine, par le biais d'un compilateur. Nous allons donc, dans un premier temps voir les parties essentielles d'un ordinateur. Ensuite, pour entrer progressivement dans le vif du sujet, nous essaierons un premier petit programme, afin de fixer les diverses étapes de l'édition et de la compilation.

1.1 L'ordinateur

Un ordinateur est constitué d'un processeur (CPU, pour Central Processor Unit) relié aux différents composants : la mémoire centrale (ou mémoire vive) et la mémoire de masse (disque dur), ainsi que les périphériques d'entrée-sortie, le clavier et l'écran (voir figure 1.1).

La mémoire de masse sert à stocker des informations à long terme (même lorsque l'ordinateur n'est plus sous tension), tandis que la mémoire vive stocke les informations

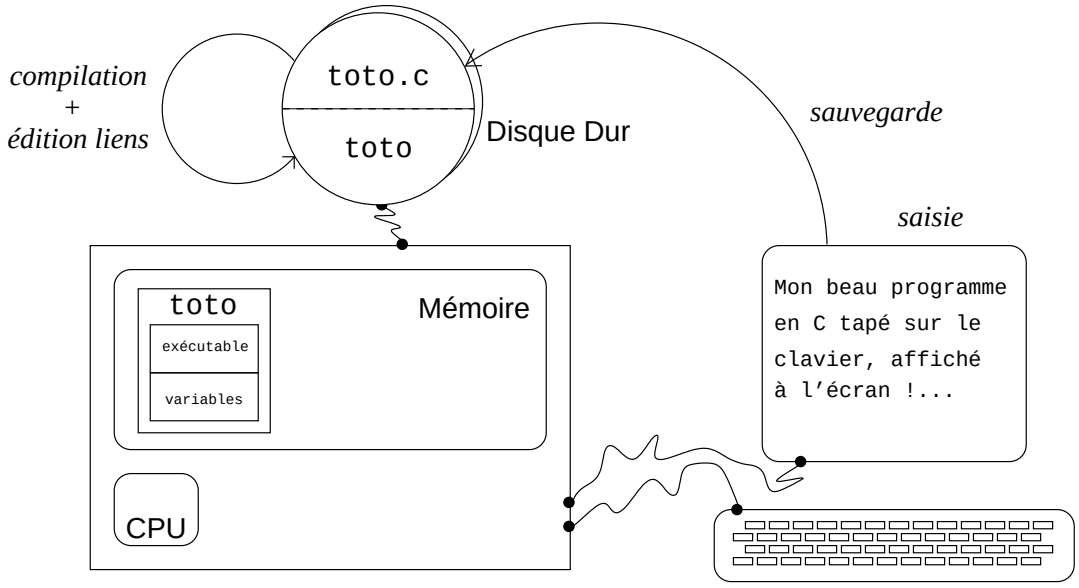

FIG. 1.1 – Organisation d'un ordinateur – Programmation en C

nécessaires à la bonne exécution d'une tâche ponctuelle : le programme (la liste des instructions à exécuter), les variables, etc. Les informations stockées en mémoire vive seront perdues à l'extinction de l'ordinateur.

L'utilisateur dialogue avec l'ordinateur par l'intermédiaire du clavier (entrée standard) et de l'écran (sortie standard).

1.2 Programmer en C

1.2.1 Généralités

L'utilisateur tape son programme en langage C, avec un *éditeur de texte*, et sauvegarde le *fichier source* sur le disque dur. Afin de le rendre compréhensible par l'ordinateur, il le *compile* en langage machine, avec un *compilateur C*, puis *édite les liens*, produisant ainsi une version *exécutable*.

Le programme *exécutable* peut alors être envoyé au processeur : l'*exécutable* est chargé en mémoire centrale. Le processeur exécute alors l'ensemble des instructions, utilisant la mémoire centrale pour stocker ses calculs temporaires.

Pour comprendre toutes ces étapes, décrites plus loin plus en détail, considérons notre premier programme, le classique " Hello World ! ! " (voir programme 1). Il faut tout d'abord le saisir au clavier, le compiler (préprocesseur, assembleur puis optimiseur de code), éditer les liens, puis l'exécuter (voir figure 1.2).

1.2.2 Un éditeur de texte : `emacs`

La première étape consiste à saisir et à sauvegarder le programme source `hello.c`. Pour cela, nous utiliserons l'éditeur de texte bien connu `emacs` : à l'invite du shell, on tape

hello.c

```
/* Mon premier programme en C */
#include <stdio.h>
int main ()
{
  printf("Hello World !!\n");
  return 0;
}
```

Programme 1: Hello World (`hello.c`)

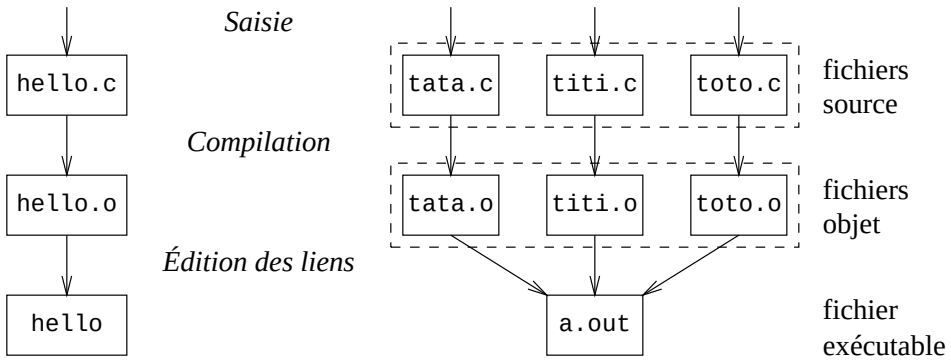

FIG. 1.2 – Les étapes de la programmation

> `emacs hello.c &`

Le " & " en fin de ligne permettant de récupérer la main dans le shell après avoir lancé l'éditeur, afin de pouvoir compiler, sans quitter l'édition en cours, qui sera sans doute encore nécessaire pour la correction des erreurs repérées par le compilateur (ne vous affolez pas, une compilation marche rarement du premier coup !).

Les inconditionnels de vi ou vim, kate, ou eclipse ou de tout autre éditeur pourront garder leurs habitudes. Cependant, `emacs` comme d'autres éditeurs, a de nombreux avantages lors de la programmation : il possède un mode C qui " indente " au fur et à mesure de la saisie. Cela clarifie la lecture, mais aussi permet de détecter les erreurs classiques que nous verrons plus loin (oubli d'accolades, de parenthèses, de points-virgules, etc).

De plus, `emacs` est un éditeur très puissant et très pratique. Il peut manipuler plusieurs fichiers à la fois dans différents " buffers ", sans lancer plusieurs `emacs` en parallèle (voir le menu déroulant **buffer** pour la liste des fichiers ouverts). Enfin, il offre une interface conviviale au débogueur (voir la section 10.3). Les commandes essentielles peuvent être trouvées dans les menus déroulants, mais des raccourcis-clavier existent (voir figure 1.3) qui permettent de limiter l'usage de la souris.

Fichier	
Ctrl-x Ctrl-f	ouvrir un fichier
Ctrl-x Ctrl-s	sauver le fichier sous le nom courant
Ctrl-x Ctrl-w	sauver le fichier sous un nouveau nom
Buffer	
Ctrl-x Ctrl-b	afficher la liste des buffers ouverts
Ctrl-x b	changer de buffer courant
Ctrl-x k	fermer un buffer
Ctrl-x 2	couper la fenêtre en 2, verticalement
Ctrl-x 3	couper la fenêtre en 2, horizontalement
Ctrl-x 1	une seule fenêtre
Ctrl-x Ctrl-w	sauver le fichier sous un nouveau nom
Déplacement	
Ctrl-a	début de ligne
Ctrl-e	fin de ligne
Début	début du fichier
Fin	fin du fichier
Édition	
Ctrl-<SPC>	marquer un début de bloc
Ctrl-w	couper du début du bloc jusqu'à la position actuelle
Alt-w	copier du début du bloc jusqu'à la position actuelle
Ctrl-y	coller le bloc copié ou coupé
Ctrl-_	annuler la dernière commande (puis les précédentes, etc)
Divers	
Ctrl-g	annuler toute commande en cours tout dialogue dans le mini-buffer
Ctrl-x Ctrl-c	fermer tous les buffers et quitter

FIG. 1.3 – Raccourcis-clavier sous `emacs`

1.2.3 Analyse du programme

Avant de " lancer " la compilation, analysons brièvement ces quelques lignes. Tout d'abord, `hello.c` est le nom du *fichier source*. Un programme est une liste de fonctions. Le programme en question n'en contient qu'une seule, appelée `main`, en raison de sa simplicité. Mais dès qu'un programme se complique, il ne faut pas hésiter à multiplier les fonctions.

1.2.3.1 Les fonctions

`int main()` est l'*en-tête* de la fonction, dont le *corps* est décrit entre les accolades qui suivent. Comme l'indique son nom, la fonction `main` est la fonction principale, la seule à être appelée lors de l'exécution du programme. C'est donc à elle de distribuer les tâches, si de multiples tâches doivent être effectuées. Il ne s'agit pas pour autant de surcharger le corps de cette fonction. En effet, il est conseillé d'écrire des fonctions spécifiques pour chaque tâche et sous-tâche, ces fonctions pouvant alors être appelées les unes par les autres, et par la fonction `main` en particulier, voire s'appeler elles-mêmes lors de fonctions dites " récursives ". Nous verrons cela plus en détail dans le chapitre 7.

Ce programme étant très simple, nous n'avons pas multiplié les fonctions, mais il faut garder à l'esprit que plus les fonctions seront simples et courtes, plus elles auront de chance d'être correctes.

L'en-tête d'une fonction (ou " prototype ") décrit ses caractéristiques, vues de l'extérieur :

- entre parenthèses, on liste les arguments que la fonction prend en entrée, avec leur type. La fonction `main` ne prend aucun argument, d'où les parenthèses vides " () ";
- à la gauche de l'en-tête, est précisé le type de sortie de la fonction. Pour la fonction `main`, il s'agit d'un entier retourné au shell appelant, correspondant au code d'erreur (0, lorsque tout s'est bien passé, une autre valeur signifie une erreur).

Le corps d'une fonction décrit le mode opératoire, ce que doit faire la fonction. Pour ce qui est de la fonction `main` étudiée, le corps est en effet élémentaire :

- la ligne " `printf("Hello World !!\n");` " affiche à l'écran la chaîne de caractères entre guillemets. Le `\n` correspond à un saut de ligne.
- finalement, la ligne " `return 0;` " conclut la fonction, retournant l'entier de sortie de la fonction, le code d'erreur 0 signifiant que tout s'est bien passé.

1.2.3.2 Les commentaires

Un programme écrit un jour sera peut-être relu ou utilisé ensuite par quelqu'un d'autre, ou par vous-même dans quelques années. Sans commentaires sur le pourquoi et le comment de ceci et de cela, la lecture sera ardue. Ainsi est-il conseillé d'ajouter des commentaires dans vos programmes. Ils ne surchargeront pas l'exécutable ni ne ralentiront son exécution : les commentaires sont supprimés à la compilation.

La ligne " `/* Mon premier programme en C */` ", comme tout texte entre `/*` et `*/` est un commentaire.

1.2.3.3 Le préprocesseur

La ligne “ `#include <stdio.h>` ” est interprétée par le préprocesseur. Il insère alors le fichier `stdio.h` à cet emplacement avant d'envoyer le programme au compilateur.

Ce fichier contient les en-têtes des fonctions usuelles pour les entrées-sorties (STanDard Input-Output), telle que `printf`.

De nombreuses directives peuvent être ainsi données au préprocesseur, telles que des insertions conditionnelles, des macros, etc. Toutes ces directives apparaissent impérativement en début de ligne (donc une seule par ligne), en commençant par un #.

1.2.4 Un compilateur C : `gcc`

Afin de traduire le programme C en langage machine, on fait appel à un compilateur. Nous utiliserons `gcc` (pour GNU C Compiler). Un tel compilateur effectue en fait plusieurs tâches successives :

- le préprocesseur, qui, comme on l'a déjà vu ci-dessus, interprète quelques commandes élémentaires pour compléter et mettre en forme le fichier source ;
- le compilateur, en tant que tel, qui traduit le fichier source en code assembleur, un langage très proche du langage machine ;
- l'optimiseur de code, qui remplace certaines séquences d'instructions par d'autres plus efficaces, déroule certaines boucles, supprime certains sauts, etc. L'optimisation est très spécifique aux caractéristiques du processeur ;
- l'assembleur, qui traduit le code assembleur en langage machine. On obtient alors le *fichier objet* ;
- l'éditeur de liens, qui interface les différents fichiers objets entre eux, mais également avec les bibliothèques usuelles fournies avec la distribution classique du C.

En général, toutes ces opérations sont effectuées en une seule étape, transparente pour l'utilisateur (avec `gcc hello.c`). On obtient alors un exécutable dénommé `a.out`, que l'on peut exécuter depuis le shell :

```
> gcc hello.c
> a.out
```

-E	lance le préprocesseur (mais n'effectue aucune compilation) > `gcc -E hello.c` retourne le résultat sur la sortie standard
-S	création du fichier assembleur (avant l'édition des liens) > `gcc -S hello.c` produit `hello.s`
-c	création du module objet (avant l'édition des liens) > `gcc -c hello.c` produit `hello.o`
-O2/-O3	optimisation du code
-o <output>	permet de spécifier le nom du fichier créé
-v	mode *verbose* qui explique tout ce qu'il fait
-Wall	compilation “ exigeante ”, informant des erreurs bénignes

FIG. 1.4 – Options du compilateur `gcc`

Diverses options peuvent être utilisées lors de la compilation (voir figure 1.4), notamment pour contrôler les phases de la compilation que l'on veut effectuer. Les plus utiles

étant " `-c` " lors d'un projet faisant intervenir plusieurs modules qui nécessitent des compilations séparées, et " `-o` " qui permet de préciser le nom du fichier exécutable (dénommé par défaut `a.out`). L'option " `-Wall` " est également très utile car elle demande au compilateur d'informer l'utilisateur de toute anomalie dans le source. Il s'agit souvent d'erreurs d'inadvertences, non fatales à la compilation, mais qui rendent l'exécution parfois incorrecte.

1.3 Quelques généralités sur la syntaxe du langage C

1.3.1 Mise en page et indentation

Un programme C peut être écrit très librement, à l'exception des directives envoyées au préprocesseur qui commencent nécessairement par un # et en début de ligne (et donc une seule directive par ligne). À part cela, les sauts de lignes et les espaces multiples ne sont pas significatifs pour le compilateur.

Reste maintenant à écrire un programme qui aide à la compréhension, et notamment au déboguage. Pour cela, l'éditeur de texte `emacs` est d'un précieux recours. En effet, il aide à indenter correctement (et peut même colorer les différentes mots selon leur caractéristique). Si cela n'est pas automatique, taper
`Alt-x c-mode` (pour passer dans un environnement C)
`Alt-x global-font-lock-mode` (pour passer en mode coloré)

Il est alors fortement conseillé de respecter les règles d'indentation : décaler d'un cran (ex : trois espaces) vers la droite tout bloc inclus dans un précédent. On peut ainsi mieux repérer qui dépend de quoi, et si tous les blocs ouverts ont bien été fermés.

1.3.2 Les identificateurs

Les noms de variables, de constantes ou de fonctions sont composés de caractères au choix parmi les lettres `a-z` et `A-Z`, et les chiffres `0-9`, avec la possibilité d'utiliser le symbole `_` (par exemple, pour remplacer l'espace).

Cependant, ces noms ne peuvent pas commencer par un chiffre. De plus, les minuscules/majuscules sont différenciées par la plupart des compilateurs.

1.3.3 Les instructions

Les instructions peuvent être de deux types :
- les instructions simples, toujours terminées par un " `;` ", tel que
  ```
  printf("Hello World !!\n");
  ```
- les instructions composées, qui sont des listes d'instructions simples encadrées par des accolades, tel que
  ```
  {
    printf("Hello World !!\n");
    printf("Bonjour !!\n");
  }
  ```

Remarque : toutes les instructions simples se terminent par un point-virgule. Son oubli est l'erreur la plus courante !

1.3.4 Organisation d'un programme C

Un programme C est alors un assemblage de fonctions, elles-mêmes listes de blocs d'instructions. Ainsi, un programme C aura toujours l'aspect présenté sur le programme 2. C'est-à-dire, outre les commentaires et les directives pour le préprocesseur qui peuvent être insérés à tout moment, on commence par déclarer les variables globales (qui devront être connues par tous les éléments du programme). Ensuite, on définit toutes les fonctions, en particulier la fonction `main`.

Chaque fonction est définie par son en-tête, puis un corps. Le corps d'une fonction commence, comme le programme, par déclarer toutes les variables nécessaires à la fonction, qui ne seront connues qu'à l'intérieur de la fonction (variables locales). On initialise ensuite ces variables, puis on liste les instructions que la fonction doit effectuer.

La fonction `main`, avec l'en-tête `int main (int argc, char *argv[])` sera plus courante que le version abrégée du programme 1. En effet, elle permet de saisir des valeurs sur la ligne de commande au moment de l'exécution du programme :

```
> gcc prog.c -o prog
> prog 14 25
```

Ainsi, les chaînes de caractères `"prog"`, `"14"` et `"25"` sont placées dans le tableau `argv`, dans les cases de 0 à la valeur de `argc` $-$ 1. Tout ceci étant effectué par le shell, vous disposez donc de ces variables pré-remplies.

Ensuite, il suffit de placer le `"14"` (contenu dans `argv[1]`) en tant qu'entier (la conversion s'effectue avec la commande `atoi`, pour " ASCII to Integer ") dans la variable `a`. On fait de même avec le `"25"` dans la variable `b`.

Enfin, on appelle la fonction `ma_fonction` sur ces variables. Grâce à ce petit programme, il est très facile de se familiariser avec les notions de base du langage C.

Nous allons pouvoir maintenant rentrer un peu plus dans les détails, en voyant notamment les différents types que le langage C sait manipuler, puis ensuite les commandes simples et structurées qui sont à votre disposition pour élaborer un programme capable de faire des calculs laborieux à notre place.

generique.c

```
/* Commentaires à tout endroit                          */
/* Directives pour le préprocesseur
   en-tête de fichier, mais également à tout endroit */
#include <stdlib.h>
#include <stdio.h>

/* Déclaration des variables globales                   */
int ma_variable;

/* Des fonctions avec en-tête et corps */
  /* en-tête de la fonction */
int ma_fonction(int x, int y)
  /* corps de la fonction   */
{
  /* Déclaration des variables locales à la fonction */
  int z;
  /* Liste d'instructions                               */
  z = x + y;
  /* Renvoi de la valeur de retour de la fonction    */
  return z;
}

float ma_deuxieme_fonction()
{
  float x=0;
  /* ... */
  return x;
}

/* La fonction obligatoire: main                        */
int main (int argc, char *argv[])
{
  /* Déclaration des variables locales à la fonction */
  int a,b,c;
  /* Initialisation des variables                       */
  a = atoi(argv[1]);
  b = atoi(argv[2]);
  c = ma_fonction(a,b);
  printf("%d + %d = %d \n",a,b,c);
  return 0;
}
```

Programme 2: Programme générique (`generique.c`)

Chapitre 2

Les types

Sommaire

Nous avons déjà aperçu le type `int` dans le programme 1, et entre-aperçu le type `float` et les tableaux dans le programme 2, mais le langage C connaît de nombreux autres types très utiles. Les plus simples étant les entiers, les réels et les caractères. Le langage C permet également de manipuler des types structurés, tels que les tableaux (ensembles d'objets de même type) ou les structures (ensembles d'objets de types divers).

2.1 Les types

Un ordinateur stocke toutes les données sous le même format, une série de bits, dans des octets (blocs de 8 bits) localisés par une adresse mémoire. Ainsi, un entier, une lettre ou un réel doivent-ils subir un codage spécifique avant d'être stockés.

De plus, connaître les bits contenus dans un octet à une adresse donnée ne suffit pas à connaître la valeur " réellement " stockée. En effet, cette valeur dépendra du type de l'objet que l'on a stocké : par exemple, le caractère `'A'` et l'entier `65` sont tous deux codés de la même manière, le type de l'objet permet de déterminer le codage et donc le contenu réel.

Le langage C sait coder un certain nombre de types usuels, nous allons les étudier dans ce chapitre.

2.2 Les types de base

Commençons donc par les types les plus simples et les plus utiles, à savoir les entiers, les flottants et les caractères.

2.2.1 Les types entiers (`int`, `short int`, `long int` et `long long int`)

Les entiers sont codés à l'aide de la représentation des nombres entiers relatifs : un bit est réservé pour coder le signe, les autres bits codent la valeur absolue du nombre. Le langage C peut gérer plusieurs tailles différentes d'entiers :

- `short int` ou simplement `short`
- `long int` ou simplement `long`
- `long long int` ou simplement `long long`
- `int` (l'entier par défaut)

Cependant, ces tailles ne sont pas standardisées, seuls des seuils minimums sont imposés. C'est-à-dire que ces tailles dépendent de la machine, du système d'exploitation et du compilateur. Avec `gcc` sous Linux sur un PC, le type `short` code des entiers sur 16 bits, tandis que les types `int` et `long` codent des entiers sur 32 bits et enfin le type `long long` code des entiers sur 64 bits. Ces tailles peuvent être trouvées dans le fichier `limits.h`, ou obtenues par l'intermédiaire de l'instruction C `sizeof(<type>)`.

À titre indicatif, avec 16 bits, on peut représenter des entiers entre -32 768 et 32 767, avec 32 bits, on couvre les valeurs de -2 147 483 648 à 2 147 483 647, puis avec 64 bits, les valeurs s'étendent de -9 223 372 036 854 775 808 à 9 223 372 036 854 775 807.

Remarque : ces trois types admettent le qualificatif `unsigned`. Dans ce cas, il n'y a plus de bit pour représenter le signe, et seuls les nombres positifs sont codés. On gagne alors un bit : avec 16 bits, on peut représenter des entiers non-signés entre 0 et 65 535, avec 32 bits, on couvre les valeurs de 0 à 4 294 967 295, puis sur 64 bits, les valeurs s'étendent de 0 à 18 446 744 073 709 551 615.
Cependant, l'emploi de ce qualificatif doit être limité à des situations particulières.

Type	Espace
`short`	[-32 768, 32 767]
`int/long`	[-2 147 483 648, 2 147 483 647]
`long long`	[-9 223 372 036 854 775 808, 9 223 372 036 854 775 807]
`unsigned short`	[0, 65 535]
`unsigned int/long`	[0, 4 294 967 295]
`unsigned long long`	[0, 18 446 744 073 709 551 615]

FIG. 2.1 – Types entiers sous GCC/Linux

2.2.2 Les types flottants (`float`, `double` et `long double`)

Les types flottants codent les nombres réels de manière approchée avec la notation " scientifique ", une mantisse et l'exposant en base 2 (ex. 234×2^{45}).

Le C propose trois tailles pour représenter ces flottants, avec une mantisse et un exposant de tailles variables :

- `float`
- `double`, le flottant en double précision
- `long double`, un type récent qui propose une précision quadruple.

Comme pour les entiers, les tailles et précisions des flottants ne sont pas standardisées, mais peuvent être trouvées dans le fichier `float.h`. Avec `gcc` sous Linux sur un PC, nous avons :

- pour les `float`, une mantisse sur 24 bits et un exposant sur 8 bits, proposant une précision de l'ordre de 10^{-7} ;
- pour les `double`, une mantisse sur 53 bits et un exposant sur 11 bits, proposant une précision de l'ordre de 2×10^{-16} ;
- pour les `long double`, une mantisse sur 64 bits et un exposant sur 16 bits, proposant une précision de l'ordre de 10^{-19}.

Type	Mantisse	Précision	Min	Max
`float`	24	10^{-7}	10^{-38}	$3 \cdot 10^{38}$
`double`	53	$2 \cdot 10^{-16}$	$2 \cdot 10^{-308}$	10^{308}
`long double`	64	10^{-19}	10^{-4931}	10^{4932}

FIG. 2.2 – Types flottants sous GCC/Linux

2.2.3 Le type caractère (`char`)

Les caractères sont essentiels, notamment pour écrire du texte, manipuler des lettres, etc. En C, ils sont représentés comme des entiers, sur un octet (entre 0 et 255) en utilisant la table de conversion ASCII. Cependant, cette table de conversion est transparente pour l'utilisateur, en notant le caractère entre apostrophes “ ’ ’ ”. Ainsi, la notation `'A'` désignera le caractère “ A ”.

2.3 Les types structurés

Un type semble manquer dans la liste ci-dessus, il s'agit des chaînes de caractères, c'est-à-dire les mots ou les phrases. En fait, elles sont représentées comme des tableaux de caractères. De nombreuses commandes standard permettent de manipuler ces tableaux particuliers de caractères (voir la section 11.1).

2.3.1 Les tableaux

Les tableaux permettent de stocker une grande quantité d'objets de même type. Ainsi, les instructions du programme 3 déclarent des tableaux de 10 entiers, de 10 flottants ou de 10 caractères (chaîne de caractères). Les cases se dénommeront alors `tab_i[0]` jusqu'à `tab_i[9]` (pour le tableau d'entiers).

tableaux.c

```
int tab_i[10];
float tab_f[10];
char tab_c[10];
```

Programme 3: Tableaux (`tableaux.c`)

2.3.2 Les tableaux à plusieurs dimensions

Il est possible de définir des tableaux à plusieurs dimensions. Il s'agit en fait d'un tableau de tableaux :

```
int tab[10][5];
```

est un tableau de 10 tableaux de 5 entiers chacuns. Ainsi, `tab[0]` désigne le premier tableau, et `tab[0][0]` la première case de ce premier tableau.

2.3.3 Les structures

Les structures, quant à elles, permettent de stocker plusieurs objets de types divers. On définit une structure comme présenté dans le programme 4.

structure.c

```
struct ma_structure {
  int nombre;
  float valeur;
};
```

Programme 4: Structure (`structure.c`)

Le type `struct ma_structure` regroupe alors deux champs, un entier et un flottant. Un objet `st` de type `struct ma_structure` contient ces deux valeurs dans ses deux champs dénommés `st.nombre` et `st.valeur`.

L'exemple d'une structure pour manipuler des complexes est donné dans la section 7.3.

2.4 Les pointeurs

Ce type un peu particulier pourrait également être classé dans les types de base, mais les pointeurs ne semblent sans doute pas naturels, ni essentiels, au premier coup d'œil, et

pourtant nous y consacrerons tout un chapitre (voir chapitre 8).

En effet, comme nous l'avons vu précédemment, tout objet est stocké en mémoire. Son contenu est donc caractérisé par l'adresse mémoire de ce stockage et du type de l'objet. Un pointeur contient cette adresse. Sa taille est donc la taille nécessaire pour coder une adresse mémoire (en général 32 bits). Un pointeur est de plus, en général, associé à un type d'objet, pour pouvoir manipuler l'objet correctement (ce que l'on verra plus loin). Ainsi, un pointeur sur un objet `obj` de type `int` contient l'adresse où est stocké l'objet `obj`. Son type est noté " `int *` ".

2.5 Nouveaux types

Les tableaux ou les structures permettent de faire des tableaux de tableaux de structures de tableaux, etc. On obtient alors des types avec des noms " à dormir dehors "!! La commande `typedef` permet de définir des synonymes pour des noms de types. Ainsi, par exemple, la structure définie ci-dessus est de type `struct ma_structure`. Si on veut définir un tableau de taille 20 d'une telle structure, son type sera alors `struct ma_structure[20]`, etc. On peut alors simplifier ce nom final, s'il doit être beaucoup utilisé. Par exemple, dans le programme 5, on renomme le nouveau type de tableau de 20 structures en `tabs`. Puis, un tel tableau dénommé `T` est aisément déclaré.

On peut aussi souhaiter aussitôt renommer le nouveau type résultant de l'instruction `struct new_structure`. Le programme 5 définit ce nouveau type sous le nom `point2D`. On peut même se passer du nom temporaire, comme le montre la définition du type `point3D` du même programme.

typedef.c

```
typedef struct new_structure {
  float abscisse;
  float ordonnee;
} point2D;

typedef struct {
  float abscisse;
  float ordonnee;
  float hauteur;
} point3D;

struct ma_structure {
  int nombre;
  float valeur;
};
typedef struct ma_structure[20] tabs;

point2D p2;
point3D p3;
tabs T;
```

Programme 5: Typedef (`typedef.c`)

Chapitre 3

Les variables et les constantes

Sommaire

Un programme informatique sert à travailler sur des valeurs, tel que faire des calculs sur des entiers ou des flottants. Ces valeurs doivent alors être stockées en mémoire. On utilise alors des *variables* si elles doivent varier. En revanche, ces variables sont en général initialisées par des valeurs *constantes*.

Les variables doivent être *déclarées*, pour spécifier le type de la variable en question. Cette déclaration attribut de plus un emplacement mémoire à la variable, qui lui est réservé pendant toute *sa durée de vie*, désigné par le pointeur de la variable. Il faut ensuite *initialiser* la variable. En effet, lors de la déclaration, un emplacement mémoire est attribué à la variable, sans toucher au contenu, qui peut donc valoir n'importe quoi. Pour cela, nous aurons alors à faire aux constantes, dont la syntaxe est spécifique au type.

3.1 Déclaration des variables

La déclaration d'une variable consiste donc, dans un premier temps, à préciser au compilateur le type de la variable que l'on va utiliser, et donc le mode de codage qu'il va devoir effectuer pour le stockage. Ensuite, en fonction du type de la variable, il va réserver une place mémoire plus ou moins importante (selon qu'il s'agit d'un `short` ou d'un `long`).

3.1.1 Déclaration des types simples

Ces déclarations s'effectuent simplement en donnant le type de la variable suivi du nom de la variable en question :

```
int i;
float x;
double y;
char c;
```

Nous avons alors déclaré une variable entière de type `int` dénommée `i`, ainsi que deux variables flottantes, une de type `float` dénommée `x` et une de type `double` dénommée `y`, et enfin une variable de type `char` dénommée `c`.

Lorsque l'on souhaite déclarer plusieurs variables du même type, on peut regrouper la déclaration sur une seule ligne :

```
int i,j,k;
float x,y;
```

3.1.2 Déclaration des tableaux et pointeurs

Formellement, les tableaux et les pointeurs ne sont pas identiques, mais nous n'insisterons pas sur les différences. Pour un tableau ou un pointeur, la déclaration est alors aussi simple, sachant que la taille du tableau doit être une constante, et non une variable :

```
float *pfloat;
int tab[10];
char st[];
```

Ainsi, nous venons de déclarer un pointeur `pfloat` sur un `float`, ainsi qu'un tableau `tab` de 10 entiers. Ces deux déclarations ont du même coup réservé l'espace mémoire nécessaire pour stocker le pointeur `pfloat` ainsi que le tableau `tab` et ses dix cases.

En revanche, la déclaration du tableau de caractères `st` n'a pas réservé de mémoire pour ses cases, puisque la longueur est inconnue. Seule la zone mémoire pour stocker l'**adresse** du futur tableau est réservée, ainsi `st` n'est rien d'autre qu'un pointeur, qui ne pointe sur rien. Lorsque la zone mémoire aura été réservée (grâce à la commande `malloc` que nous étudierons dans le chapitre 8 dédié aux pointeurs), nous pourrons alors définir cette valeur. Cette méthode est la seule pour manipuler des tableaux dont la taille n'est pas une constante connue au moment de la compilation.

En revanche, `tab` est un pointeur de type `int` qui pointe sur le début de l'espace mémoire réservé pour les 10 cases du tableau (voir la figure 3.1). La première case du tableau est donc à cette adresse, la deuxième case, à cette adresse incrémentée de la taille d'un entier (ce que l'on connaît, puisque le pointeur est typé en tant que pointeur sur un `int`), etc.

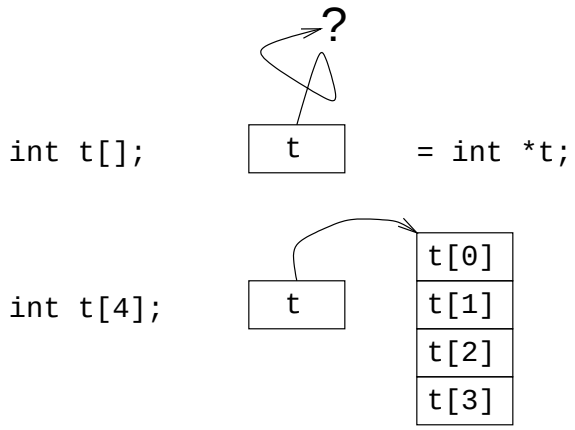

FIG. 3.1 – Déclaration des tableaux

3.1.3 Déclaration des objets complexes

Finalement, pour tout type, tel que le type " tableau de structures " `tabs` défini à la fin du chapitre précédent, on déclare une variable comme la variable `T` du programme 5. La taille d'un tel objet étant bien définie, un tableau de 20 cases contenant chacune un entier et un flottant, la mémoire est également réservée. Ainsi `T` est un pointeur vers une zone mémoire qui contient 20 cases consécutives d'objets de type `struct ma_structure`.

En revanche, considérons le type et la déclaration suivante :

```
typedef struct {
  int length;
  int value[];
} bigInteger;

bigInteger bI;
```

La variable `bI` est bien déclarée comme étant de type `bigInteger`. Elle contient donc un champ de type entier, `bI.length`, et un champ de type tableau d'entiers, sans longueur. Pour ce dernier, il s'agit donc d'un pointeur sans destination précise (voir la figure 3.2). Il faudra donc l'initialiser ultérieurement, lorsqu'un espace mémoire aura été réservé (grâce à la commande `malloc` que nous verrons plus loin dans le chapitre 8 dédié aux pointeurs).

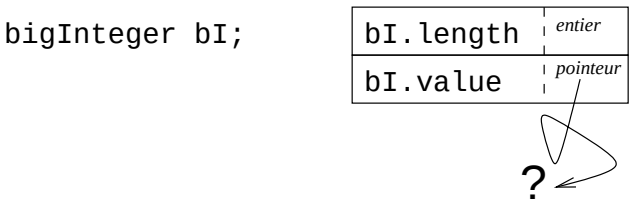

FIG. 3.2 – Déclaration des structures

3.2 Initialisation des variables

La déclaration des variables se contente de signaler le type de la variable et de réserver, le cas échéant, de la mémoire. Dans ce dernier cas, la réservation de mémoire ne modifie pas son contenu précédent. Ainsi, la variable prend la valeur associée au contenu de la mémoire : ça peut être n'importe quoi. Il faut donc toujours initialiser une variable avant de l'utiliser. Ainsi,

```
int i;
i = 0;
```

déclare une variable entière `i`, puis l'initialise à 0. La déclaration et l'initialisation peuvent être faites en une seule étape au lieu de deux :

```
int i = 0;
```

En fait, l'initialisation consiste à affecter une valeur constante à la variable. La syntaxe des constantes est décrite dans la section suivante.

3.3 Constantes

3.3.1 Types simples

Les constantes pour les types simples ont une syntaxe particulière, étant donné qu'il existe plusieurs types d'entiers et de flottants, puis qu'il existe également plusieurs notations.

3.3.1.1 Constantes de type entier

Les entiers sont construits à partir des chiffres (et des lettres pour une notation en hexadécimale). On ajoute alors éventuellement un suffixe " l " ou " L " pour désigner une constante de type `long` et/ou le suffixe " u " ou " U " pour désigner le qualificatif `unsigned`. Aucun préfixe signale que le nombre est exprimé en base 10, tandis que les préfixes `0` et `0x` précisent respectivement une notation en octale ou en hexadécimale.

Ainsi, `0377` désigne l'entier 255, `0xF` désigne l'entier 15. En revanche, `120L` désigne l'entier 120 de type `long`.

3.3.1.2 Constantes de type flottant

Les flottants peuvent bien sûr être désignés par des nombres à virgule (enfin, avec un point " . "!). On ajoute éventuellement un suffixe " f " ou " F " pour désigner une constante de type `float` ou le suffixe " l " ou " L " pour désigner le type `long double`. On peut également utiliser la notation scientifique `123E+3` ou `-142E-4`.

3.3.1.3 Constantes de type caractère

Les caractères sont simplement désignés entre apostrophes. Ainsi, la constante `'a'` désigne le caractère *a minuscule*. On peut également noter ces lettres par leur code ASCII en utilisant les notations octales ou hexadécimales, précédées d'un backslash. Ceci permet de noter les caractères non affichables. Certains caractères admettent des abbréviations :

`'\101'` et `'\x41'`	désignent le caractère A, de code ASCII 65, (également `'A'`)
`'\0'` et `'\x0'`	désignent le caractère nul (délimite la fin des chaînes de caractères)
`'\12'` et `'\xA'`	désignent le saut de ligne (également `'\n'`)
`'\13'` et `'\xB'`	désignent le retour chariot (également `'\r'`)

3.3.2 Types complexes

Pour les types complexes (tableaux et structures), on initialise chaque case ou champ indépendamment, un par un. On peut également en initialiser plusieurs à la fois :

`int inttab[5] = { 2, 5, -2, 0, 10 };`	tableau de 5 entiers contenant les valeurs 2, 5, -2, 0 et 10
`float floattab[4] = { 2.0, 5e-3, -2e10, 10.04 };`	tableau de 4 flottants contenant les valeurs 2.0, 5×10^{-3}, -2×10^{10} et 10.04
`char mot[5] = "essai";`	tableau de 5 caractères 'e', 's', 's', 'a', 'i'
`int tab[10] = { 2, 5, -2, 10 };`	tableau de 10 entiers dont les 4 premières cases sont initialisées à 2, 5, -2 et 10
`char *mot = "essai";`	pointeur sur un caractère, initialisé à l'adresse de la chaîne `"essai"`
`char chaine[10] = "mot";`	tableau de 10 caractères, les premières cases sont initialisées à `'m'`, `'o'`, `'t'` et `'\0'`
`struct carte { char[] nom; int age; } prem = { "durand", 10 };`	définition du type `struct carte` déclaration d'une variable `prem` initialisation avec `prem.nom="durand"` et `prem.age=10`

3.4 Durée de vie et visibilité des variables

Les variables ont une durée de vie plus ou moins longue, ainsi qu'une visibilité variable. Voir les programmes 6 et 7 pour un exemple d'application. Les fichiers `variables.c` et `options.c` sont deux modules de compilation différents, le fichier `variables.c` étant l'unité principale, puisqu'elle contient la fonction `main`. On les compile d'un seul coup de la façon suivante :

```
>gcc options.c variables.c -o variables
>variables 4
10 + 4 = 14
```

- **globale** — les variables définies en dehors de toute fonction sont dites *globales*. Elles sont vues/accessibles par toutes les fonctions (au moins du module). Elles ont une durée de vie égale au temps d'exécution du programme. Telles les variables `b` et `c` définies en tête du fichier `variables.c`.
- **locale** — les variables définies dans une fonction ou dans un bloc sont *a priori* visibles et existantes dans le seul bloc en question. Telles les variables `b` et `d` déclarées en tête de la fonction `main`. Elles n'existeront que dans cette fonction. Si une telle variable a un nom identique à une variable globale, la variable globale ne sera pas visible ni accessible dans cette zone, mais seule la variable locale. Telle la variable locale `b` définie en tête de la fonction `main` qui " masque " la variable globale du même nom.
- `extern` — il s'agit d'un qualificatif qui permet d'utiliser dans une unité de compilation une variable ou un type définis dans un autre module. Ainsi, la variable `a` définie dans l'unité `options.c` est annoncée comme utile dans l'unité principale `variables.c`.
- `static` — il s'agit d'un qualificatif qui modifie la visibilité de la variable. Une variable globale `static` ne pourra pas être utilisée par une autre unité de compilation (voir le qualificatif `extern`). Une variable locale `stactic` aura une durée de vie égale à l'exécution du programme mais ne sera visible que dans le bloc (ou la fonction) dans lequel elle a été définie.
- `register` — ce qualificatif informe le compilateur que la variable sera très utilisée, et lui demande de la mettre, si possible, dans un registre du processeur.

Ces deux derniers qualificatifs sont d'un usage limité. Notamment l'utilisation des registres. En effet, le nombre de registres est très faible, et il vaut mieux faire confiance au compilateur, et notamment à l'optimiseur de code, pour les optimisations de ce genre.

variables.c

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

extern int a;
int b = 0;
int c;

int main (int argc, char *argv[])
{
  int b;
  int d;
  b = atoi(argv[1]);
  c = a+b;
  d = a*b;
  printf("%d + %d = %d \n",a,b,c);
  printf("%d * %d = %d \n",a,b,d);
  return 0;
}
```

Programme 6: Variables (`variables.c`)

options.c

```
int a=10;
```

Programme 7: Variables – annexe (`options.c`)

Chapitre 4

Les entrées-sorties

Sommaire

On a vu comment passer des arguments par l'intermédiaire de la ligne de commande. Ils se trouvent stockés dans le tableau `argv[]` de longueur `argc`, les arguments de la fonction `main` : `int main(int argc, char *argv[])`. Il est également possible que le programme pose des questions à l'utilisateur et lise les données entrées au clavier. Ceci est effectué avec la fonction `scanf`. Nous décrirons donc son usage. Cependant, elle est moins utile que la fonction `printf` qui permet d'écrire des données à l'écran.

Le programme 8 présente un exemple d'utilisation des fonctions `printf` et `scanf`, dont l'exécution donne l'écran suivant :

```
>gcc io.c -o io
>io
Entrer deux entiers : 12 76
Entrer deux flottants : 3.54 9.234
   12 *    76 =   912
 3.54 +  9.23 = 12.77
```

4.1 L'affichage avec `printf`

4.1.1 Simple affichage

La fonction `printf` sert à afficher des données à l'écran. Les chaînes de caractères constantes s'affichent telles que, mais les autres constantes et les variables doivent être intégrées de façon particulière, spécifique au type et au format d'affichage.

io.c

```
#include <stdio.h>

int main ()
{
  int a,b,c;
  float x,y,z;

  printf("Entrer deux entiers : ");
  scanf("%d %d",&a,&b);

  printf("Entrer deux flottants : ");
  scanf("%f %f",&x,&y);

  c = a*b;
  z = x+y;
  printf("%5d * %5d = %5d \n",a,b,c);
  printf("%5.2f + %5.2f = %5.2f \n",x,y,z);
  return 0;
}
```

Programme 8: Entrées/Sorties (`io.c`)

Ainsi, `printf("Affichage");` écrit le mot *Affichage* à l'écran. Pour afficher le contenu d'une variable entière, on spécifie `%d` (voir le programme 2) pour préciser qu'il s'agit d'un type entier, puis on précise le nom de la variable à insérer en fin :

`printf("La variable i vaut %d",i);`

Si on veut afficher la variable en octal ou en hexadécimal, on utilise `%o` et `%x` respectivement. Pour les flottants, on utilise `%f` ou `%e` selon que l'on souhaite un affichage standard ou en notation exponentielle. Une liste plus complète peut être trouvée figure 4.1 pour afficher les entiers, les flottants et les chaînes de caractères.

La fonction `printf` retourne un entier qui peut être récupéré ou non. Cet entier vaut le nombre de caractères affichés (sans le caractère de fin de chaîne de caractères).

4.1.2 Formattage des sorties

Un formattage plus complexe est possible avec cette commande, notamment pour préciser le nombre d'espaces que l'on prévoit pour l'affichage d'un entier, pour former des colonnes :

`printf("%4d et %5d",i,j);`

affiche les variables entières `i` et `j` en insérant, devant, des espaces de façon à ce que les espaces plus l'affichage du `i` (resp. du `j`) prennent 4 caractères (resp. 5 caractères).

Type et Variable	Affichage	Format
`short i; int j; long k;`	`printf("%hd %d %ld",i,j,k);`	décimal
`short i; int j; long k;`	`printf("%ho %o %lo",i,j,k);`	octal
`short i; int j; long k;`	`printf("%hx %x %lx",i,j,k);`	hexadécimal
`float x; double y;`	`printf("%f %lf",x,y);`	décimal (avec point)
`float x; double y;`	`printf("%e %le",x,y);`	exponentiel
`float x; double y;`	`printf("%g %lg",x,y);`	le plus court des deux
`long double z;`	`printf("%Lf",z);`	décimal (avec point)
`long double z;`	`printf("%Le",z);`	exponentiel
`long double z;`	`printf("%Lg",z);`	le plus court des deux
`char c;`	`printf("%c",c);`	caractère
`char m[10];`	`printf("%s",m);`	chaîne de caractères

FIG. 4.1 – Affichage des variables avec `printf`

```
  3 et    9
 12 et  235
2353 et 4532
```

Pour les flottants, il est de même possible de préciser le nombre de chiffres après la virgule, ainsi que le nombre total de caractères (point compris) :

```
printf("%6.2f %.3Lf",x,y);
```

affiche la variable flottante `x` avec 2 chiffres après la virgule, mais en insérant des espaces devant afin que le tout fasse 6 caractères. Tandis que le variable de type `long double y` sera simplement affichée avec 3 chiffres après la virgule, sans spécification de la longueur totale.

Enfin, il est également possible de formatter l'affichage des chaînes de caractères en précisant la taille globale (le nombre d'espaces à insérer en tête) :

```
printf("%20s",chaine);
```

affiche la chaîne de caractère `chaine` avec des espaces devant afin que le tout fasse 20 caractères.

4.2 La saisie avec `scanf`

La fonction `scanf` permet de saisir des valeurs tapées au clavier pendant l'exécution du programme. La syntaxe est très semblable à celle de la fonction `printf`, mais en sens inverse : `scanf("valeurs %d et %d",&i,&j);` attend, au clavier, une phrase de la forme *valeurs 10 et 31* et mettra la valeur *10* dans la variable `i` et la valeur *31* dans la variable `j`.

On remarquera que la notation des variables diffère de leur utilisation avec `printf`. En effet, il faut indiquer à la fonction `scanf` l'adresse de la variable dans laquelle on veut stocker la valeur scannée, d'où le `&` qui accède au pointeur associé à la variable qui suit (voir le chapitre 8 sur les pointeurs).

Type et Variable	Affichage	Format

`short i;`
`int j;`
`long k;`
	`scanf("%hd",&i);`
	`scanf("%d",&j);`
	`scanf("%ld",&k);`
		décimal

`short i;`
`int j;`
`long k;`
	`scanf("%ho",&i);`
	`scanf("%o",&j);`
	`scanf("%lo",&k);`
		octal

`short i;`
`int j;`
`long k;`
	`scanf("%hx",&i);`
	`scanf("%x",&j);`
	`scanf("%lx",&k);`
		hexadécimal

`float x;`
`double y;`
`long double z;`
	`scanf("%f",&x);`
	`scanf("%lf",&y);`
	`scanf("%Lf",&z);`
		décimal (avec point)

`float x;`
`double y;`
`long double z;`
	`scanf("%e",&x);`
	`scanf("%le",&y);`
	`scanf("%Le",&z);`
		exponentiel

`char c;`
`char m[10];`
	`scanf("%c",&c);`
	`scanf("%s",m);`
	`scanf("%s",&m[0]);`
		caractère
		chaîne de caractères

FIG. 4.2 – Saisie avec `scanf`

La fonction `scanf` retourne un entier, qui peut être récupéré ou non :

```
k = scanf("valeurs %d et %d",&i,&j);
```

la valeur retournée, stockée dans `k`, vaut le nombre de variables scannées. En effet, si l'utilisateur tape *valeurs 10 est 31*, le *10* sera lu, mais ne voyant pas le mot *et* apparaître, rien d'autre n'est lu. Ainsi la fonction `scanf` retourne 1. Si l'utilisateur tape n'importe quoi, la fonction `scanf` retourne 0. Pour une utilisation normale, telle que la phrase entrée ci-dessus, la fonction `scanf` retournera 2.

Chapitre 5

Les opérateurs et les expressions

Sommaire

Maintenant que l'on connaît les variables, les constantes, comment les définir et les afficher, voyons comment les manipuler, les modifier avec les opérations standard.

5.1 Les opérateurs

Les opérateurs sont des commandes qui s'appliquent à des opérandes, qui sont des constantes, des variables ou des expressions (voir la section suivante).

Les opérateurs peuvent être répertoriés en fonction du nombre d'opérandes qu'ils prennent comme arguments : les opérateurs unaires, qui ne prennent qu'un argument (voir figure 5.1), les opérateurs binaires, qui prennent deux arguments, tels que les opérateurs arithmétiques classiques (addition, soustraction, etc). Une liste plus complète est proposée figure 5.2. Enfin, l'opérateur ternaire, qui prend trois arguments. Commençons par ce dernier type, qui ne contient qu'un cas dans le langage C :

```
test ? expr1 : expr2
```

Le premier argument est un test `test`, en fonction de son état de validité, le résultat de l'expression `expr1` ou de l'expression `expr2` est retourné : `expr1` pour un test vrai, `expr2` pour un test faux. Ceci est une version abrégée des instructions de condition (`if` et `else`) que nous verrons dans le chapitre suivant. À éviter pour produire un programme lisible !

Au lieu d'étudier les autres opérateurs plus classiques par leur nombre d'opérandes, regroupons-les par catégories de procédures.

5.1.1 Les opérateurs arithmétiques

5.1.1.1 Opérateurs standard

Les opérations arithmétiques se font avec les opérateurs classiques : +, -, * et /. Si les arguments sont des entiers, les opérations retournent des entiers. Notamment, la division " `/` " retourne le quotient de la division euclidienne, tandis que " `%` " retourne le reste de cette division euclidienne.

Sinon, les quatre opérateurs de base manipulent des flottants.

`arithmetique.c`

```
int main ()
{
  int a,b,c;
  float x,y,z;

  a = 5; b = 7;
  c = b/a;       /*  alors c = 1 */
  c = b%a;       /*  alors c = 2 */
  c = b++;       /*  alors b = 8 et c = 7 */
  c = ++a;       /*  alors a = 6 et c = 6 */
  c = a >> 3;    /*  alors c = 48 */
  c = b << 2;    /*  alors c = 2 */

  x = 3.01; y = 0.02;
  z = -x*y;      /*  alors z = -0.0602 */

  return 0;
}
```

Programme 9: Opérateurs arithmétiques (`arithmetique.c`)

5.1.1.2 Opérateurs d'incrémentation ++ et −−

Il existe des opérateurs d'incrémentation et de décrémentation abbrégés :
`i++;` et `++i;` reviennent à écrire `i = i+1;`
`i--;` et `--i;` reviennent à écrire `i = i-1;`

Catégorie	Opérateur	Action
arithmétique	$--$	décrémentation
	$++$	incrémentation
	$-$	inverser le signe
	$+$	confirmer le signe
type	`(type)`	convertit le type de la variable (“ cast ”)
adressage (pointeurs)	`&`	retourne l'adresse (pointeur) d'un objet
	`*`	retourne le contenu d'une variable à l'adresse indiquée
logique	`!`	NOT logique
masquage	`~`	NOT bit à bit
divers	`sizeof`	taille de l'objet (ou du type) en octets

FIG. 5.1 – Opérateurs unaires

Catégorie	Opérateur	Action
arithmétique	$+$	addition
	$-$	soustraction
	$*$	multiplication
	$/$	division (entière si entre entiers)
	$\%$	reste de la division euclidienne
décalages	$\ll$	vers la gauche
	$\gg$	vers la droite
logique	&&	ET logique
	`||`	OU logique
masquage	`&`	ET bit à bit
	`|`	OU bit à bit
	`^`	XOR bit à bit
relations	`<`	inférieur strict
	`<=`	inférieur ou égal
	`>`	supérieur strict
	`>=`	supérieur ou égal
	`==`	égal
	`!=`	différent
divers	`=`	affectation

FIG. 5.2 – Opérateurs binaires

Cependant, les valeurs retournées par ces opérations diffèrent :
- `j = i++;` est une post-incrémentation de `i`. C'est-à-dire que `j` prend la valeur de `i`, puis `i` est incrémenté ;
- `j = ++i;` est une pré-incrémentation de `i`. C'est-à-dire que `i` est incrémenté, puis `j` prend la valeur du `i`, après incrémentation.

Il en est de même pour les `i--;` et `--i;` (voir le programme 9).

5.1.1.3 Opérateurs de signe

Les opérateurs unaires + et − permettent de confirmer on d'inverser le signe :
`j = +i ; k = -i ;`

5.1.1.4 Opérateurs de décalage

Les opérateurs de décalage peuvent être assimilés à des opérateurs arithmétiques, bien que leur usage standard soit pour des masquages. Mais, en fait, un décalage de n à droite revient à une division par 2^n, tandis qu'un décalage à gauche de n revient à une multiplication par 2^n.

5.1.2 Opérateur de conversion de type (“ cast ”)

Si des scalaires de différents types interviennent dans une expression arithmétique, le type final de l'expression est le type le plus général des divers éléments : les types plus stricts sont automatiquement “ castés ” dans le type plus général (voir la figure 5.3). On

`short` → `int` → `long` → `float` → `double`→ `long double`

FIG. 5.3 – Cast automatique

peut également forcer la conversion de type, en mettant le nouveau type entre parenthèses devant la variable :

```
int a = 5;
int b = 7;
float c,d;

c = a/b;
d = (float) a/b;
```

Dans le programme ci-dessus, la variable `c` prend le résultat de la division euclidienne (entre entiers), donc `0`. Tandis que la variable `d` prend la résultat de la division entre l'entier `a` converti en `float` et l'entier `b`. Ainsi, le résultat est un `float` : `0.714286`.

5.1.3 Opérateur de taille

Puisque nous somme dans les types, nous avons vu qu'il pouvait être utile de connaître la taille d'un objet, pour savoir la place nécessaire en mémoire. Nous verrons encore plus

l'utilité pour l'allocation dynamique de mémoire, dans le chapitre sur les pointeurs (voir chapitre 8).

La commande `sizeof` fournit cette information, en nombre d'octets : `sizeof(int)` retourne la taille d'un entier, c'est-à-dire 4 octets pour `gcc` sous linux. De même, si `a` est une variable de type `int`, `sizeof(a)` retourne la taille de cette variable, et donc la taille d'un `int`, soit 4 octets.

5.1.4 Opérateurs logiques

Il est possible de manipuler des valeurs booléennes, représentées par des entiers : le `0` valant *faux*, tandis que toute valeur non nulle représente le *vrai* (en général, on utilise simplement le `1`). Ainsi, les opérateurs `!`, `&&` et `||` représentent respectivement le NOT, ET et OU logiques.

5.1.5 Opérateurs de masquage

Il est également possible de manipuler indépendamment chaque bit d'un entier, un à un, avec des opérations logiques. Ceci permet notamment de masquer certains bits, mais également d'en inverser, etc.

Ainsi, les opérateurs `~`, `&`, `|` et `^` représentent respectivement le NOT, ET, OU et XOR (OU exclusif) effectués bit à bit.

5.1.6 Opérateurs de relations

Pour comparer certaines valeurs, on peut utiliser les relations binaires classiques, d'égalité mais également d'ordre strict ou non :

- le test d'égalité est effectué avec l'opérateur `==` tandis que le test de différence est effectué avec l'opérateur `!=`.
- les comparaisons strictes sont effectuées avec les opérateurs `<` et `>`.
- les comparaisons larges sont effectuées avec les opérateurs `<=` et `>=`.

Chacun de ces tests retourne un booléen (codé sous forme d'entier, comme vu précédemment).

5.1.7 Opérateur d'affectation

Comme on vient de le voir, le test d'égalité s'effectue avec l'opérateur binaire `==`. En effet, l'opérateur simple `=` est l'opérateur d'affectation et non de test d'égalité. Il affecte la valeur de droite à la variable de gauche :

```
int a,b;

a = 7;
b = a + 9;
```

Si on souhaite affecter la même valeur à plusieurs variable, il est possible de les affecter en cascade : `i = j = k =10 ;` qui affecte la valeur `10` aux variables `i`, `j` et `k`.

Remarque : Ne pas confondre les opérateurs = et ==. C'est une erreur classique, souvent difficile à détecter, car le compilateur ne peut s'en rendre compte, et ne le signale donc pas !

5.1.8 Opérateurs abrégés

De même que les opérateurs d'incrémentation informent le compilateur que la variable elle-même doit être incrémentée, il existe des versions abrégées de tous les opérateurs arithmétiques et de masquage qui informent le compilateur que l'opération doit s'effectuer sur la variable elle-même (voir la figure 5.4).

Catégorie	Opérateur	Action
arithmétique	`a += 5 ;`	`a = a + 5 ;`
	`a -= 7 ;`	`a = a - 7 ;`
	`a *= 2 ;`	`a = a * 2 ;`
	`a /= 3 ;`	`a = a / 3 ;`
	`a %= 3 ;`	`a = a % 3 ;`
décalage	`a <<= 5;`	`a = a << 5;`
	`a >>= 7;`	`a = a >> 7;`
masquage	`a &= 0x10 ;`	`a = a & 0x10 ;`
	`a |= 077 ;`	`a = a | 077 ;`
	`a ^= 0xFF ;`	`a = a ^ 0xFF ;`

FIG. 5.4 – Opérateurs abrégés

5.2 Les expressions

Une expression est une suite d'opérateurs et d'opérandes.

Chapitre 6

Les structures de contrôle

Sommaire

Un programme C est une suite séquentielle d'opérations. Il est bien sûr possible de conditionner l'exécution de certaines opérations au résultat de calculs préalables, ou de répéter un nombre fini (ou infini) de fois un groupe d'opérations.

6.1 Instruction

Dans la suite, nous utiliserons le terme *instruction* pour désigner soit une *instruction simple*, terminée par un point-virgule, soit un *groupe* d'instructions simples, encadré par des accolades.

6.2 Les boucles

Les boucles permettent de répéter des instructions un grand nombre de fois. Il existe trois syntaxes différentes en fonction des besoins.

6.2.1 Les boucles `while`

La syntaxe générale du `while` est

```
while (<test>)
  <instruction>
```

Cela signifie que tant que le test `<test>` est satisfait, l'instruction `<instruction>` est exécutée. Si le test est " faux " dès le début, l'instruction n'est jamais exécutée. Un exemple classique d'utilisation de la boucle `while` est le calcul du PGCD avec la méthode d'Euclide (voir programme 10).

pgcd.c

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

int main (int argc, char *argv[])
{
  int a,b,r,x,y;
  x = a = atoi(argv[1]);
  y = b = atoi(argv[2]);

  while (b != 0)
    {
      r = a % b;
      a = b;
      b = r;
    }
  printf("pgcd (%d,%d) = %d\n",x,y,a);
  return 0;
}
```

Programme 10: PGCD (`pgcd.c`)

6.2.2 Les boucles `for`

La syntaxe générale du `for` est

```
for (<initialisation>; <test>; <incrementation>)
  <instruction>
```

qui est équivalente à

```
<initialisation>
while (<test>)
  {
    <instruction>
    <incrementation>
  }
```

Une syntaxe plus classique est de la forme

```
for (i = 0 ; i < n ; i++)
  <instruction>
```

qui consiste à effectuer n fois l'instruction avec des valeurs croissantes pour i. La boucle `for` est souvent utilisée avec les tableaux (voir programme 11).

carres.c

```
#include <stdio.h>
#define N 10

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

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

  for (i=1; i<N; i++)
    t[i] = t[i-1] + t[i];

  for (i=0; i<N; i++)
    printf("%3d -> %3d \n",i,t[i]);
  return 0;
}
```

Programme 11: Carrés (`carres.c`)

6.2.3 Les boucles `do while`

La syntaxe générale du `do while` est

```
do
  <instruction>
while (<test>)
```

Cette commande est parfois connue dans d'autres langages (tel en PASCAL) en `repeat – until`. Elle est très semblable à la boucle `while`, sauf que l'instruction `<instruction>` est exécutée au moins une fois. Un exemple d'utilisation de la boucle `do while` est présentée ci-dessous (voir programme 12).

binaire.c

```
#include <stdlib.h>
#include <stdio.h>
#define N 32

int main (int argc, char *argv[])
{
  int x,a,i,j;
  int t[N];
  a = atoi(argv[1]);
  x = a;
  i = 0;

  do
    {
      t[i] = a % 2;
      a = a/2;
      i++;
    }
  while (a != 0);

  printf("%d = ",x);
  for (j = i-1; j>=0; j--)
    printf("%d",t[j]);
  printf("\n");
  return 0;
}
```

Programme 12: Binaire (`binaire.c`)

6.3 Les conditions

Les exécutions conditionnelles peuvent être de plusieurs types. Le test simple `if ... else ...`, ou la table de branchement `switch/case`.

6.3.1 Condition `if`

La commande de test simple peut être une condition

```
if <test>
  <instruction>
```

où l'instruction `<instruction>` est exécutée seulement si le test `<test>` est satisfait, ou une alternative

```
if <test>
  <instruction1>
else
  <instruction2>
```

où l'instruction `<instruction1>` est exécutée si le test `<test>` est satisfait, sinon c'est l'instruction `<instruction2>` qui est effectuée, tel que dans le programme de parité (voir programme 13).

parite.c

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

int main (int argc, char *argv[])
{
  int a;
  a = atoi(argv[1]);

  if (a%2 == 0)
    printf("nombre pair \n");
  else
    printf("nombre impair \n");
  return 0;
}
```

Programme 13: Parité (`parite.c`)

6.3.2 Condition `switch`

Lorsque les alternatives sont plus nombreuses, on peut utiliser la commande `switch` qui désigne la valeur à étudier, suivie des instructions à exécuter en fonction des divers cas (commande `case`). Plus précisément, l'exécution des commandes commence lorsque la valeur étudiée apparaît, jusqu'à la fin (ou jusqu'au `break`). Si la valeur n'apparaît jamais, le cas `default` englobe toutes les valeurs :

```
switch (a)
  {
    case 1: <instruction1>
    case 2: <instruction2>
            break;
    case 4: <instruction4>
            break;
    default: <instruction>
  }
```

Ainsi, si `a` vaut `1`, le programme est branché sur l'instruction `<instruction1>`, suivie de l'instruction `<instruction2>`. La commande `break` interrompt l'exécution pour la poursuivre après l'accolade. Si `a` vaut `2`, le programme est branché sur l'instruction `<instruction2>`. La commande `break` interrompt l'exécution pour la poursuivre après l'accolade. Si `a` vaut `4`, le programme est branché sur l'instruction `<instruction4>`. La commande `break` interrompt l'exécution pour la poursuivre après l'accolade. Enfin, dans tous les autres cas, on exécute l'instruction `<instruction>`.

6.4 Sauts Inconditionnels

Il existe en C des sauts pour quitter des boucles ou pour aller n'importe où dans une fonction. Nous allons juste les citer par soucis d'exactitude, mais ils sont à bannir d'un programme bien structuré (sauf éventuellement pour gérer des erreurs/exceptions).

6.4.1 `break`

Nous avons vu cette commande en conjonction avec la commande `switch`. C'est le seul usage " propre ". En effet, il permet de sortir d'un bloc, en sautant automatiquement à l'instruction qui suit l'accolade fermante.

6.4.2 `continue`

Dans une boucle, la commande `continue` fait passer à l'itération suivante (sans exécuter les instructions suivantes dans le bloc).

6.4.3 `return`

Comme on va le voir dans le chapitre suivant, la commande `return` termine l'exécution d'une fonction. Si la fonction est de type `void`, on utilise la commande `return` seule. Si

valeur.c

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

int main (int argc, char *argv[])
{
  int a = atoi(argv[1]);

  switch (a)
    {
    case 0: printf("Positif \n");
    case 2: printf("Plus petit que 3 \n");
    case 4: printf("Plus petit que 5 \n");
    case 6: printf("Pair ! \n");
      break;
    case 1: printf("Positif \n");
    case 3: printf("Plus petit que 3 \n");
    case 5: printf("Plus petit que 5 \n");
    case 7: printf("Impair ! \n");
      break;
    default: printf("Plus grand que 8 ! \n");
    }
  return 0;
}
```

Programme 14: Valeur (`valeur.c`)

la fonction a un autre type, elle doit retourner une valeur (par exemple `a`) à la fonction appelante, avec un `return a`.

6.4.4 `goto`

Comme dans les autres langages, le C permet des sauts inconditionnels n'importe où à l'intérieur d'une fonction. Il permet ainsi de sortir de plusieurs niveaux de blocs (ce que les fonctions `break` et `continue` ne permettent pas). Le `goto` est associé à une étiquette, une chaîne de caractères suivie de " : ".

```
{
  {
    {
      ...
      if (negatif) goto erreur ;
    }
  }
}
erreur : printf("Log d'un nombre négatif !\n");
```

Chapitre 7

Programme Structuré

Sommaire

Comme nous l'avons vu dans l'introduction, pour améliorer la lisibilité, un programme doit être modulaire. Pour cela, nous allons découper le programmes en divers modules, qui eux-mêmes contiennent plusieurs fonctions qui s'entre-appellent. Le programme contient une (seule) fonction principale, la fonction `main`, celle qui est appelée au lancement du programme.

Nous allons d'abord étudier de plus près les fonctions, puis nous verrons comment découper un programme en plusieurs modules.

7.1 Les fonctions

Les fonctions permettent d'écrire, une fois pour toutes, une série d'instructions qui pourront être exécutées plusieurs fois, sans avoir à tout réécrire. Une fonction doit avoir un nom explicite, et une en-tête complète (ou prototype), afin que la seule vue de cette en-tête permette de savoir comment utiliser la fonction. Ensuite, est décrit le mode opératoire de cette fonction.

7.1.1 Déclaration d'une fonction

Une fonction est déclarée grâce à son en-tête (*interface* ou *prototype*). Le compilateur se contentera de cette interface pour effectuer la première phase de la compilation : il lui suffit de savoir ce que prend la fonction en entrée et ce qu'elle retourne (les types de ces éléments). Le contenu, le mode opératoire de la fonction lui importe peu.

```
int addition (int x, int y) { <corps> }
void affichage (int t[], int n) { <corps> }
```

Ainsi, l'en-tête de la fonction `addition` précise que cette fonction prend deux entiers comme arguments (ils seront appelés `x` et `y` dans le corps de la fonction), puis qu'elle retourne un entier.

L'en-tête de la fonction `affichage` précise que cette fonction prend un tableau d'entiers (un tableau de taille quelconque qui sera dénommé `t` dans le corps de la fonction), ainsi qu'un entier (qui sera dénommé `n` dans le corps de la fonction et sera supposé désigner la taille effective du tableau) comme arguments, puis ne retourne rien. En effet, une telle fonction *affichera* des résultats à l'écran, mais ne *retournera rien* à la fonction appelante.

7.1.2 Corps de la fonction

Dans la partie `<corps>`, le corps de la fonction, est décrit le mode opératoire. C'est-à-dire que l'on donne la liste des fonctions à exécuter. Ainsi, voici le contenu du corps de la fonction d'affichage d'un tableau :

```
void affichage (int t[], int n)
{
  int i;
  for (i=0; i<n; i++) printf("%3d ",t[i]);
  printf("\n");
}
```

7.1.3 Retour de la réponse

La fonction `affichage` ne retournait aucun résultat, elle se contentait d'afficher le contenu du tableau, passé en argument, à l'écran. En revanche, certaines fonctions effectuent des calculs à retourner à la fonction appelante. Ce résultat est alors retourné à l'aide de la commande `return`.

```
int addition (int x, int y)
{
  int z;
  z = x + y;
  return z;
}
```

7.1.4 Arguments d'une fonction

Il faut bien voir les arguments des fonctions comme des " variables muettes ". Par exemple, pour appeler la fonction `affichage` sur un tableau `s` de longueur `l`, on écrit `affichage(s,l)`. Alors le contenu de la variable `l` est copié dans une variable locale à la fonction, appelée `n`. De même, le contenu de la variable `s` (le pointeur vers la zone mémoire qui contient le tableau) est copié dans une variable locale à la fonction, appelée `t`.

De même, pour obtenir la somme de deux entiers `a` et `b`, on appelle `addition(a,b)`. Pour stocker le résultat retourné par la fonction dans une variable entière `c`, déjà déclarée, on écrit `c = addition(a,b)`.

Alors le contenu de la variable `a` est copié dans une variable locale à la fonction, appelée `x`. De même, le contenu de la variable `b` est copié dans une variable locale à la fonction, appelée `y`. C'est l'ordre des arguments qui permet de savoir le nom qu'ils auront au sein de la fonction.

Comme la fonction manipule des copies des variables passées comme argument, si la fonction modifie les variables locales, les variables initiales demeurent inchangées. Voir le programme 15.

incrementation.c

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

int incrementation(int x)
{
  x++;
  return x;
}

int main (int argc, char *argv[])
{
  int a,b;
  a = atoi(argv[1]);

  b = incrementation(a);
  printf("a = %d, b = %d\n",a,b);
  return 0;
}
```

Programme 15: Incrémentation (`incrementation.c`)

Dans ce programme, la variable `a` (locale à la fonction `main`) prend la valeur passée sur la ligne de commande (par exemple 10). Lors de l'appel de la fonction `incrementation`, son contenu est copié dans une variable `x`, locale à la fonction `incrémentation`. Cette

variable `x` a donc, initialement, la même valeur que `a`, c'est-à-dire 10. Elle est incrémentée, alors `x` vaut désormais 11, mais `a` n'a pas été modifiée. La fonction `incrémentation` retourne la valeur de `x` à la fonction appelante (la fonction `main`), qui la stocke dans la variable `b`. Ainsi, dans cette fonction `main`, la variable `a` est inchangée, et vaut 10. La variable `b` a pris la valeur retournée par la fonction `incrementation`, et vaut donc 11 :

```
>incrementation 10
a = 10, b = 11
```

On parle alors de passage *par valeur*. Seule la valeur de la variable est transmise à la fonction, qui ne peut donc modifier cette variable. Nous verrons que les pointeurs permettent des passages *par adresse*, afin que la fonction puisse modifier le contenu d'une variable (voir chapitre 8).

7.1.5 Fonction `main`

La fonction `main` est particulière. En effet, c'est la seule fonction appellée automatiquement lors du lancement du programme. Le programme doit donc en contenir une, et une seule. Ainsi, son en-tête est-elle imposée par le shell. Son interface est

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

Comme toujours, les arguments sont des variables muettes. Ainsi, `argc` et `argv` peuvent prendre des noms différents, mais les types sont fixés. Le shell passe d'ailleurs des arguments bien précis :

- l'argument `char *argv[]` désigne un tableau de chaînes de caractères. Ce tableau contient donc plusieurs mots, tous les mots passés sur la ligne de commande au moment du lancement du programme. L'espace servant de séparateur. Ainsi, en lançant
  > `incrementation 10`
  La première case du tableau (`argv[0]`) contient `incrementation`, la deuxième (`argv[1]`) contient `10`.
- l'argument `int argc` désigne la longueur du tableau (le nombre de mots sur la ligne de commande), soit 2.

7.2 Récursivité

Comme on l'a vu avec la fonction `main` qui appelle la fonction `incrementation`, toute fonction peut appeler toute autre fonction, à partir du moment où les types des arguments sont cohérents. Il n'est d'ailleurs pas nécessaire d'avoir déjà défini la fonction que l'on appelle, il suffit que son prototype soit connu pour l'utiliser. Il faudra cependant qu'elle soit complètement définie quelque part au moment du " link ", la dernière phase de la compilation. Ainsi, les fichiers d'en-tête, tel que `stdio.h` contiennent uniquement les entêtes des fonctions usuelles, sans les corps.

Par conséquent, rien n'empêche une fonction de s'appeler elle-même, c'est ce qu'on appelle une fonction récursive. Cette méthode de programmation sera souvent préférée, en raison de la clarté et de la simplicité des algorithmes qu'elle met en œuvre.

Cependant, il faut faire attention à ce qu'une telle fonction ne s'appelle pas indéfiniment : comme en mathématiques, une fonction récursive doit être complètement définie. Pour cela, il faut un (ou plusieurs) cas trivial qui ne fait pas appel à la récursivité. En général, le cas 0 ou le cas 1. Puis ensuite, on définit le résultat des valeurs supérieures en fonction des valeurs inférieures. L'exemple classique est le calcul de la factorielle (voir programme 16).

fact.c

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

int factorielle(int n)
{
  if (n == 1) return 1;
  return n*factorielle(n-1);
}

int main (int argc, char *argv[])
{
  int a,b;
  a = atoi(argv[1]);
  b = factorielle(a);
  printf("factorielle(%d) = %d\n",a,b);
  return 0;
}
```

Programme 16: Factorielle (`fact.c`)

Elle définit une suite entière par la récurrence :

$$\begin{aligned} u_1 &= 1 \\ u_n &= n * u_{n-1} \text{ pour } n > 1 \end{aligned}$$

Il s'agit alors d'une fonction simplement récursive (et même avec récursivité terminale, voir le cours d'algorithmique pour plus de détails). Un autre exemple classique de fonction récursive, avec, cette fois-ci, une double récursivité, est le calcul de la suite de Fibonacci (voir programme 17).

En effet, elle définit la suite entière par la récurrence linéaire d'ordre 2 suivante :

$$\begin{aligned} v_1 &= 1 \\ v_2 &= 1 \\ v_n &= v_{n-1} + v_{n-2} \text{ pour } n > 2 \end{aligned}$$

fibo.c

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

int fibo(int n)
{
  if ((n == 1)||(n == 2)) return 1;
  return fibo(n-1)+fibo(n-2);
}

int main (int argc, char *argv[])
{
  int a,b;
  a = atoi(argv[1]);
  b = fibo(a);
  printf("fibo(%d) = %d\n",a,b);
  return 0;
}
```

Programme 17: Fibonacci (`fibo.c`)

Cette notion de récursivité sera étudiée plus en détail dans le cours d'algorithmique.

7.3 Les modules

7.3.1 Structure

Il est très utile de découper un programme en petites fonctions, afin de résoudre les différents problèmes un à un. De plus, cela permet de réutiliser, ultérieurement dans un autre programme, des fonctions déjà écrites. Pour cela, il peut être bien de regrouper par modules les fonctions de même type. Le programme suivant de manipulation des complexes est ainsi découpé en trois parties :

- la partie `complexe.h` (voir programme 18) qui est un fichier d'inclusion.
  Il contient la déclaration des nouveaux types (et notamment des structures) et des fonctions, avec seulement le prototype, et le qualificatif `extern`. En effet, le prototype des fonctions suffit au compilateur à savoir comment s'interface la fonction avec l'extérieur : le type des arguments, et le type de sortie des fonctions. On y déclare également les variables globales avec le qualificatif `extern`.
  Le second intérêt de ce fichier d'inclusion est qu'il permet d'avoir, en un coup d'oeil, l'ensembles des types/fonctions définies dans le module de définition (voir le suivant module).
- le module `complexe.c` de définition des fonctions (voir programme 19), avec par conséquent le contenu des corps des fonctions.

complexe.h

```
struct complexe_s {
  double reel;
  double imaginaire;
};

typedef struct complexe_s complexe;

extern complexe multiplication(complexe, complexe);
extern void affichage(complexe);
```

Programme 18: Déclaration des types et fonctions (`complexe.h`)

– le module principal (voir le programme 20) qui contient notamment la fonction `main`. Il fait l'interface entre l'utilisateur et les différentes fonctions.

7.3.2 Compilation

La compilation d'un tel programme peut se faire en plusieurs phases ou en une seule :

```
>gcc complexe.c complexe-main.c -o complexe
```

Ainsi, le compilateur fabrique les versions objet du fichier `complexe.c` et du fichier `complexe-main.c` puis édite les liens. Alors les appels aux fonctions sont effectivement " branchés " aux définitions. Ce qui produit enfin le fichier exécutable auquel on donne le nom `complexe`.

Lorsque les modules sont gros, et demandent beaucoup de temps à la compilation, il peut être long de tout recompiler après la modification d'un seul module. On peut alors simplement recompiler le module en question puis éditer les liens pour produire l'exécutable.

```
>gcc -c complexe.c
>gcc -c complexe-main.c
>gcc complexe.o complexe-main.o -o complexe
```

L'utilitaire `make` permettra l'automatisation de cette tâche : il ne lance la compilation que des modules modifiés depuis la dernière compilation (voir la section 10.1).

complexe.c

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

complexe multiplication(complexe a, complexe b) {
  complexe c;
  c.reel       = a.reel * b.reel - a.imaginaire * b.imaginaire;
  c.imaginaire = a.reel * b.imaginaire + a.imaginaire * b.reel;
  return c;
}

void affichage(complexe c) {
  printf("%g + i.%g",c.reel,c.imaginaire);
}
```

Programme 19: Définition des fonctions (`complexe.c`)

complexe-main.c

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

int main (int argc, char *argv[])
{
  complexe c,d,e;
  c.reel = atof(argv[1]);  c.imaginaire = atof(argv[2]);
  d.reel = atof(argv[3]);  d.imaginaire = atof(argv[4]);

  e = multiplication(c,d);    affichage(c);printf(" * ");
  affichage(d);printf(" = "); affichage(e);printf("\n");
  return 0;
}
```

Programme 20: Module principal (`complexe-main.c`)

7.4 Le préprocesseur

Comme nous l'avons vu dans l'introduction, la première étape de la compilation fait appel au préprocesseur. Ce processus sert à envoyer, au compilateur, un fichier source complet et consistant. Il fait donc un pré-traitement du fichier source.

7.4.1 Inclusions et définitions

Le préprocesseur peut inclure des fichiers ou remplacer des chaînes de caractères par d'autres :

- `#include <toto>` dit au préprocesseur d'insérer le fichier " standard " `toto`. Le mot " standard " signifiant qu'il doit aller le chercher parmi les fichiers-systèmes.
- `#include "toto"` dit au préprocesseur d'insérer le fichier `toto` qui se trouve dans le répertoire courant.
- `#define MAXIMUM 20` dit au préprocesseur de remplacer toutes les occurrences du mot `MAXIMUM` dans le programme par le mot `20`. Ceci définit donc des constantes. Il important de noter ici que ces commandes transmises au préprocesseur ne sont pas des instructions du langage C. En effet, elles ne se terminent pas par un ";". Notamment, pour la commande `#define`, il faut la décomposer en :

  `#define` *NOM fin de ligne*

  Alors, le préprocesseur remplacera toutes les occurrences de *NOM* par la fin de la ligne (y compris un éventuel signe de ponctuation). Une erreur dans cette " *fin de ligne* " sera signalée, par le compilateur, aux occurrences de *NOM*, et non pas à cette ligne de définition.
  On peut de la même manière définir des *macros*. Ainsi

  ```
  #define moyenne(A,B) (A+B)/2
  ```

  ---

  définit la macro `moyenne`. Toutes les occurrences de `moyenne(x,y)`, indépendamment des arguments, seront remplacées par l'expression complète du calcul de la moyenne.
  Certains motiveront cette utilisation, par rapport à l'utilisation d'une fonction, pour optimiser le code. En effet, cela ne nécessite pas d'appel à fonction, assez coûteux. Mais en contre-partie, le fichier exécutable sera plus volumineux. Ceci peut être très pénalisant : si l'exécutable est petit, il tient dans la mémoire cache du processeur est s'exécute beaucoup plus vite. Aussi, mieux vaut-il laisser l'optimiseur de code se charger de l'optimisation. En effet, les optimiseurs sont de plus en plus performants ! Ainsi, la seule motivation des macros est la lisibilité. Pour de petites opérations, les macros sont plus concises que les fonctions.
- `#undef MAXIMUM` dit au préprocesseur d'oublier une précédente définition (constante ou macro)

7.4.2 Compilation conditionnelle

Dans la phase de déboguage, ou pour permettre la compilation sur différentes plates-formes, on peut vouloir compiler telle ou telle partie en fonction de certains critères :

– en fonction d'une constante

```
#if TOTO
  part 1
#else
  part 2
#endif
```

si `TOTO` est non nulle (vraie), la partie 1 est insérée et la partie 2 supprimée. Sinon, c'est le contraire. La directive `#else` n'est pas indispensable.

– en fonction de la définition d'une constante

```
#ifdef LINUX
  part 1
#else
  part 2
#endif
```

si `LINUX` a été définie (par un `#define` par exemple, non suivi d'un `#undef` !), la partie 1 est insérée et la partie 2 supprimée. Sinon, c'est le contraire. La directive `#else` n'est pas indispensable.

– en fonction de la non-définition d'une constante

```
#ifndef MSDOS
  part 1
#else
  part 2
#endif
```

si `MSDOS` n'est pas définie (pas de `#define`, ou alors un `#undef` !), la partie 1 est insérée et la partie 2 supprimée. Sinon, c'est le contraire. La directive `#else` n'est pas indispensable.

7.4.3 Le préprocesseur `gcc`

Il est également possible de définir des constantes ou des macros sur la ligne de commande au moment de la compilation :

```
>gcc -DTOTO=10 hello.c -o hello
>gcc -DLINUX prog.c -o prog
```

Ainsi, lors de la compilation de `hello.c`, la constante/macro `TOTO` est définie à `10`. Lors de la compilation de `prog.c`, la constante/macro `LINUX` est définie (par défaut à `1`).

Chapitre 8

Les pointeurs, les tableaux et les structures

Sommaire

Les pointeurs sont une notion délicate mais indispensable à maîtriser pour utiliser correctement les structures, les tableaux, ou faire des passages par adresse dans des fonctions (afin que ces fonctions puissent modifier la variable passée en argument). De plus, ils seront à la base de tous les algorithmes et structures étudiés en algorithmique (listes, arbres, graphes, etc).

8.1 Pointeurs

8.1.1 Définition

Comme on l'a déjà vu dans le chapitre sur les types (voir chapitre 2), un pointeur contient une adresse mémoire. Cette adresse pointe alors sur on objet, qui dépend de la déclaration du pointeur. En effet, les pointeurs peuvent être typés, afin de gérer les tableaux correctement.

De plus, en passant un tel pointeur comme argument d'une fonction, il est alors possible à la fonction de modifier le contenu de la variable pointée par ce pointeur. En effet, utilisée seule, la variable passée en argument ne peut être modifiée (le pointeur) car la fonction travaille sur une copie, mais ce sur quoi elle pointe n'est pas transféré.

8.1.2 Utilisation

Il existe deux opérateurs unaires liés aux pointeurs (voir figure 5.1, page 41) :
- l'opérateur de " contenu " d'une adresse ; si `ptint` est un pointeur de type entier, `*ptint` vaut l'entier contenu à l'adresse `ptint`.
- l'opérateur d'" adresse " d'une variable ; si `a` est une variable quelconque, `&a` vaut l'adresse à laquelle est stockée la variable `a`.

8.1.3 Passage *par adresse*

Nous avons déjà eu à faire au passage par adresse dans le chapitre sur les entrées-sorties (voir chapitre 4), notamment avec la fonction `scanf`. En effet, cette fonction *scanne* une chaîne de caractères puis extrait certaines parties pour les stocker dans des variables passées comme arguments. À la sortie de la fonction, le contenu de ces variables doit réellement être la valeur introduite par la fonction, il doit rester modifié ! C'est pour cela que l'on donne, non pas la valeur de la variable à modifier, mais son adresse, en utilisant l'opérateur unaire d'adresse `&a` (voir programme 8, page 36, et figure 4.2, page 38), qui permet la modification du contenu pointé.

Ainsi, on peut modifier le programme 15 comme présenté dans le programme 21. Ce dernier modifie le contenu de la variable, dont le pointeur est passé comme argument.

auto-incrementation.c

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

void incrementation(int *ptx)
{
  (*ptx)++;
}

int main (int argc, char *argv[])
{
  int a;
  a = atoi(argv[1]);

  incrementation(&a);
  printf("a = %d\n",a);
  return 0;
}
```

Programme 21: Auto-Incrémentation (`auto-incrementation.c`)

8.2 Tableaux

Un tableau est en fait un pointeur sur une zone mémoire où sont stockées toutes les cases, de façon consécutive (voir figure 8.1). Ainsi, grâce au typage du pointeur, pour un

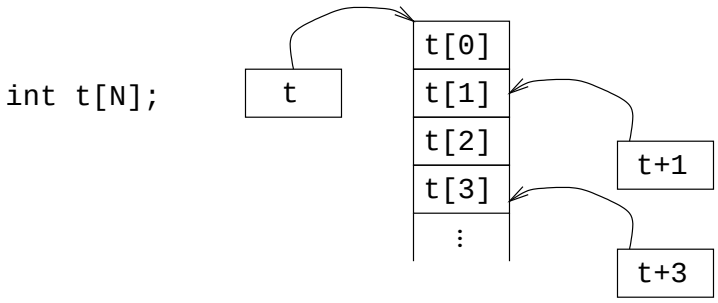

FIG. 8.1 – Tableau d'entiers

tableau `int t[N]`, de longueur `N` :

- `t` est un pointeur sur la première case du tableau, par conséquent, `t[0]` et `*t` désignent tous les deux le contenu de la première case du tableau ;
- `t+1` est donc un pointeur sur la deuxième case du tableau, par conséquent, `t[1]` et `*(t+1)` désignent tous les deux le contenu de la deuxième case du tableau ;
- `t+`i est, de façon générale, un pointeur sur la $i+1$-ième case du tableau, par conséquent, `t[`i`]` et `*(t+`i`)` désignent tous les deux le contenu de la $i+1$-ième case du tableau.

En effet, le typage du pointeur permet de définir correctement le pointeur `t+1`. Ce pointeur n'est pas l'adresse `t` plus un, mais l'adresse `t` plus la taille d'un entier.

Dans l'autre sens, il est possible d'obtenir l'adresse d'une case du tableau à l'aide de la commande `&` :

- `&t[0]` est le pointeur sur la première case du tableau. Il s'agit donc de `t` ;
- `&t[`i`]` est le pointeur sur la $i+1$-ième case du tableau. Il s'agit donc de `t+`i.
- `&t[`i`]-`j est le pointeur sur la $i-j+1$-ième case du tableau. Il s'agit donc de `t+`i`-`j.

Comme on l'a vu dans la section sur les tableaux, la déclaration d'un tableau ne peut faire intervenir qu'une constante. Ainsi, la taille d'un tableau est fixée par le programme, lors de la compilation. Heureusement, il est possible d'“ allouer ” dynamiquement la place nécessaire au stockage des cases d'un tableau.

8.3 Allocation dynamique de mémoire

Étant donné un pointeur, il est possible de le faire pointer vers une zone mémoire de taille définie dynamiquement au cours de l'exécution du programme :

- `void *calloc(size_t nmember, size_t size)` alloue la mémoire nécessaire pour un tableau de `nmember` éléments, chacun étant de taille `size` octets. Chaque case est initialisée à zéro. Il retourne le pointeur sur cette zone mémoire.
  Pour obtenir la taille de l'objet que l'on souhaite stocker dans chaque case, et comme la taille d'un entier ou d'un flottant n'est pas standardisée, on utilise la commande `sizeof`.

```
int n = 15;
double *t;
t = calloc (n, sizeof(double));
```

– `void *malloc(size_t size)` alloue `size` octets de mémoire. Son usage est semblable à celui de `calloc`, mais les octets ne sont pas initialisés (encore moins à 0).

```
int n = 15;
double *t;
t = malloc (n*sizeof(double));
```

Le type `void *` est appelé " pointeur " générique. Ce type est automatique remplacé par le type pointeur effectif dès que possible.

Le programme 22 présente une implémentation différente du calcul des éléments de la suite de Fibonacci de façon plus efficace. Elle utilise un tableau. Cependant, la taille de ce tableau n'est pas connue à l'avance puisqu'elle dépend de l'indice de l'élément de la suite que l'on cherche à calculer.

À la fin de la fonction `fibo`, la variable locale `t` disparaît, mais la mémoire allouée dynamiquement reste réservée. Ainsi, lors d'une nouvelle allocation de mémoire, cette mémoire ne sera pas considérée disponible, alors qu'elle est devenue inutilisable (le seul moyen d'y accéder était le pointeur `t`). De cette façon, très rapidement, il n'y a plus de mémoire disponible ! Pour éviter cela, on libère la mémoire qui n'est plus nécessaire avec la commande `free` exécutée sur le pointeur qui désigne l'espace mémoire en question :

```
free(t);
```

Remarque : Il est également possible d'augmenter la taille d'un tableau avec la commande `realloc`, lorsque le tableau initial s'avère trop petit. Mais cela revient à allouer de la mémoire ailleurs et à déplacer les premières cases au début de ce nouveau tableau. En effet, il n'y a aucune raison qu'à la suite du tableau initial, un espace mémoire suffisant soit disponible.

8.4 Les structures

Comme on l'a vu dans le chapitre 2 sur les types, il est possible de définir des types élaborés (voir le programme 4). On donne alors un nom plus simple à la structure et à un pointeur sur une telle structure :

fibo-tableau.c

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

int fibo(int n)
{
  int i,f;
  int *t;
  t = malloc (n*sizeof(int));
  t[0] = 1;
  t[1] = 1;
  for (i=2; i<n; i++)
    t[i] = t[i-1] + t[i-2];
  f = t[n-1];
  free(t);
  return f;
}

int main (int argc, char *argv[])
{
  int a,b;
  a = atoi(argv[1]);

  b = fibo(a);
  printf("fibo(%d) = %d\n",a,b);
  return 0;
}
```

Programme 22: Suite de Fibonacci – tableau (`fibo-tableau.c`)

```
struct ma_structure {
  int nombre;
  float valeur;
};

typedef struct ma_structure St;
typedef struct ma_structure *ptSt;
```

Le type `struct ma_structure`, ou `St`, regroupe alors deux champs, un entier et un flottant. Un objet `st` de type `St` contient ces deux valeurs dans ses deux champs dénommés `st.nombre` et `st.valeur`.

En revanche, si un objet `ptst` est un pointeur sur une telle structure, et par conséquent de type `(struct ma_structure *)`, ou simplement `ptSt`, avant de remplir les champs de la structure, il faut allouer la mémoire pour stocker cette structure :

```
ptSt ptst;
ptst = malloc(sizeof(St));
```

Enfin, le pointeur `ptst` pointe sur une telle structure qui contient donc deux champs dénommés `(*ptst).nombre` et `(*ptst).valeur`, ou plus simplement `ptst->nombre` et `ptst->valeur`.

```
ptSt->nombre = 10;
ptSt->valeur = 105.1045;
```

Un exemple, avec les listes, est présenté dans le chapitre sur les structures dynamiques (voir le programme 23, page 74).

Chapitre 9

Les structures dynamiques

Sommaire

Comme nous l'avons vu dans le chapitre précédent, les tableaux permettent de stocker de grandes quantités de données de même type. Mais, même s'il est possible d'agrandir un tableau une fois plein (ou constaté trop petit), ceci consiste en une opération coûteuse (réallocation, copie, etc). À ce genre ce structure, on préfère souvent les structures dynamiques, qui grossissent selon les besoins. Ainsi, seule la place nécessaire est réservée. Un bon exemple qui sera vu plus en détail dans le cours d'algorithmique est la structure de liste chaînée.

9.1 Structure de liste

Une liste chaînée est un *chaînage* de *cellules*. Une cellule est constituée de deux champs : le champ de valeur (appelé `val`), qui stockera les données utiles, seules vues de l'extérieur, et le champ d'adressage de la cellule suivante (appelé `next`), tel que présenté sur la figure 9.1.

La fin de la liste est désignée par une cellule particulière, appelée usuellement `nil`.

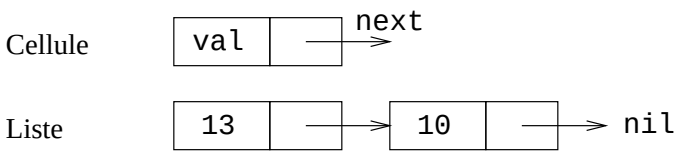

Fig. 9.1 – Liste chaînée

Cette structure de liste est très riche. C'est pour cela qu'elle est très étudiée en algorithmique. Il existe bien entendu plusieurs moyens pour l'implémenter, qui dépendent notamment du langage utilisé.

9.2 Implémentation des listes chaînées

Nous présentons, dans le programme 23, une implémentation usuelle des listes chaînées en C. Tout d'abord, la cellule est définie par la structure `struct cellule_s` que nous renommons aussitôt (par soucis de clarté) `cellule`.

Cette structure de cellule contient deux champs, un de type entier pour stocker les données utiles, puis un de type pointeur sur une cellule, pour pointer sur la cellule suivante.

Ensuite, nous définissons le type `list` en tant que tel : un pointeur sur une cellule. Puis nous désignons la fin de liste (ou la liste vide, également notée `nil`) par le pointeur `NULL`. Il s'agit d'une constante C affectée à `0`.

Une fois que le type `liste` est complètement défini, nous pouvons définir des fonctions pour agir sur ce type. Les fonctions classiques sur les listes sont :

- `list cons(int a, liste l)` qui ajoute l'élément `a` en tête de la liste `l`, et retourne la liste résultat. Pour cela, il faut créer une nouvelle cellule dans laquelle on définit la valeur à `a` et l'adresse de la cellule suivante à `l`. La liste résultante est la liste définie par le pointeur sur cette nouvelle cellule.
- `int head(list l)` qui retourne la valeur dans la cellule de tête de la liste `l`.
- `list tail(list l)` qui retourne la *queue* de la liste `l`. C'est-à-dire la liste `l` privée de sa cellule de tête.

Ces deux fonctions ne sont pas définies si `l` est la liste vide. On ne se préoccupera pas de ces cas. En effet, on pourrait gérer ces cas particuliers, appelées *exceptions*, mais le langage C ne permet pas de le faire naturellement (contrairement au langage Java). Alors, on pourrait faire un `exit(0)` pour quitter le programme dans de tels cas. L'entier passé en argument, définissant le *statut* récupéré par le shell, peut préciser le type d'erreur rencontré.

9.3 Extensions

9.3.1 Listes doublement chaînées

Une extension classique des listes chaînées est la liste doublement chaînée. En effet,

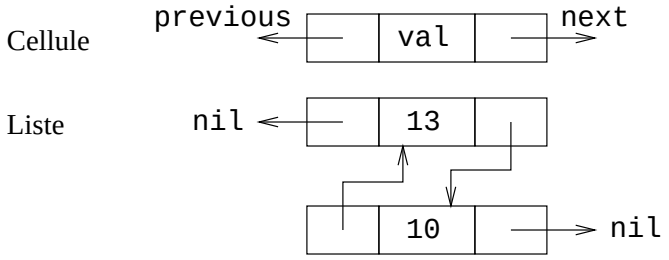

FIG. 9.2 – Liste doublement chaînée

la structure de liste simplement chaînée, comme vue dans les sections précédentes, ne permettent que de " descendre " le long de la liste. Elle implémente ainsi parfaitement la notion de *pile* (dite aussi liste LIFO, voir le cours d'algorithmique). Mais alors, lire le contenu de la dernière cellule est très long. Pour combler cette défaillance, et si aller chercher la dernière cellule est une opération courante à effectuer (telle que dans une *file* ou liste FIFO), on implémente la liste avec une liste doublement chaînée (voir la figure 9.2).

9.3.2 Arbres

Une autre extension naturelle des listes simples, où chaque cellule pointe sur la suivante, est de faire pointer une cellule sur plusieurs autres. Si chaque cellule pointe sur zéro, une ou deux cellules, on parle d'arbres binaires (voir la figure 9.3). Si le nombre est variable, on parle d'arbres généraux. Ces notions seront vues en détail dans le cours d'algorithmique. Leur implémentation fera naturellement appel aux structures (cellule, ou nœud pour les arbres) et aux pointeurs.

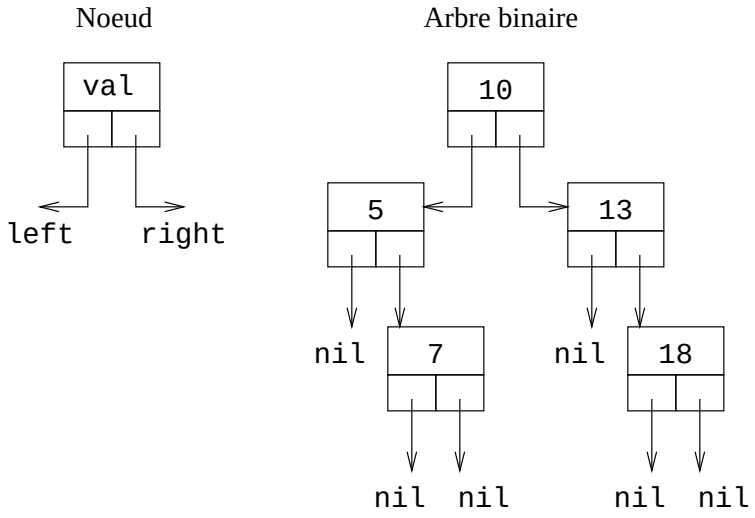

FIG. 9.3 – Arbres binaires

liste.c

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

typedef struct cellule_s {
  int val;
  struct cellule_s *next;
} cellule;
typedef cellule *list;
list nil = NULL;

list cons (int a, list l) {
  list l1;
  l1 = malloc (sizeof(cellule));
  l1->val  = a; l1->next = l;
  return l1;
}

int head (list l) {  return l->val; }

list tail (list l) { return l->next; }

void affiche(list l) {
  if (!l) printf("nil \n");
  else {
    printf("%d -> ",head(l));
    affiche(tail(l));
  }
}

int main (int argc, char *argv[])
{
  list l = nil;
  affiche(l);
  l = cons(10,l); l = cons(13,l);
  affiche(l);
  return 0;
}
```

Programme 23: Listes chaînées (`liste.c`)

Chapitre 10

L'environnement sous UNIX

Sommaire

10.1 Une construction sur mesure : `make`

Lorsqu'un programme est constitué de plusieurs modules, il n'est pas nécessaire de tout recompiler après la modification d'une seule partie. Nous avons donc vu dans la partie sur la programmation structurée, et notamment sur les modules (voir section 7.3), qu'il était possible de compiler séparément chaque module, puis ensuite éditer les liens :

```
>gcc -c complexe.c
>gcc -c complexe-main.c
>gcc complexe.o complexe-main.o -o complexe
```

Il est possible d'automatiser la re-compilation de tout (et seulement) ce qui est nécessaire, avec l'outil `make`.

Cet outil est très performant, et très complexe. Nous allons présenter les fonctions essentielles permettant de mener à bien et aisément un projet comportant plusieurs modules.

10.1.1 Fichier de Configuration : `Makefile`

La commande `make` utilise un fichier de " configuration ". Il va chercher, par défaut, dans le répertoire courant, le fichier de configuration `GNUmakefile`, `makefile` puis `Makefile`. Les deux derniers étant préférables, puis surtout `Makefile`, car il apparaîtra ainsi généralement en tête du `ls`.

Ce fichier de configuration décrit comment chaque " cible ", ou objectif à atteindre ou à créer, peut être obtenue en fonction des fichiers " sources " (initialement existants). Il précise également les " dépendances " des fichiers entre eux. Le fichier présenté programme 24 fournit un exemple, basé sur la structure suivante :

```
<cible> : <source>
<-tab-> ACTION
```

Remarque : Il est important de noter que le " blanc " devant la commande à effectuer est une " tabulation " (et non une série d'espaces).

L'exécutable `complexe` nécessite les objets `complexe.o` et `complexe-main.o` pour l'édition des liens. Chacun étant obtenu à partir du `.c` correspondant.

L'outil `make` regarde si les sources ont été modifiées à une date ultérieure à la date de création de la cible. Ainsi, si le fichier `complexe.o` est plus ancien que le fichier `complexe.c` ou `complexe.h`, ce qui signifie qu'un de ces derniers a été modifié depuis la dernière compilation, `complexe.o` sera re-compilé. Puis, récursivement, `complexe` deviendra obsolète, et sera donc reconstruit.

Makefile1

```
complexe : complexe.o complexe-main.o
        gcc complexe.o complexe-main.o -o complexe

complexe.o : complexe.c complexe.h
        gcc -c complexe.c

complexe-main.o : complexe-main.c complexe.h
        gcc -c complexe-main.c
```

Programme 24: Exemple de fichier `Makefile` (`Makefile1`)

10.1.2 Wildcards et Dépendances

Au lieu de donner la liste de tous les `.o` avec leurs `.c` et `.h` associés et la commande pour passer de l'un à l'autre, on peut simplement dire comment on obtient tout objet `.o` à partir de son `.c`.

```
%.o : %.c
    gcc -c $<
```

Dans cet exemple ; `%.o` est toute cible `.o`, produite à partir du même nom avec suffixe `.c`. Le nom exact de la source est stocké dans la variable `<` (appelée par `$<`) et le nom exact de la cible est stocké dans la variable `@` (appelée par `$@`), voir la section suivante sur les variables.

Puis on précise la liste des dépendances, c'est-à-dire la liste des fichiers dont dépendent les objets à créer, et dont les dates de modification sont à contrôler :

```
complexe.o : complexe.c complexe.h
complexe-main.o : complexe-main.c complexe.h
```

Le fichier `Makefile2` (voir programme 25) présente alors une version équivalente mais plus compacte du précédent fichier.

10.1.3 Constantes et Variables

Il est possible d'utiliser des variables dont le contenu pourra être utilisé par l'invocation de la variable :

Makefile2

```
complexe : complexe.o complexe-main.o
        gcc complexe.o complexe-main.o -o complexe

%.o : %.c
        gcc -c $<

complexe.o : complexe.c complexe.h
complexe-main.o : complexe-main.c complexe.h
```

Programme 25: Makefile : version compacte (`Makefile2`)

```
OBJETS = complexe.o complexe-main.o

complexe : $(OBJETS)
        gcc $(OBJETS) -o complexe
```

10.1.4 Cibles Fictives

Certaines cibles peuvent ne pas donner lieu à la création d'un fichier. On parle alors de " cible fictive ". Alors la commande associée sera toujours effectuée puisque `make` constate que le fichier n'existe pas. Cela est utilisé couramment pour deux actions essentielles :

– définir une action par défaut.

```
do : complexe
```

– pour donner des noms simples à des actions importantes ou souvent invoquées, tel que le nettoyage.

```
clean :
        rm -f $(OBJETS) complexe
```

10.1.5 Invocation de la Commande `make`

Au lancement de la commande `make`, sans aucun argument, l'outil cherche le fichier de configuration (`Makefile`) puis cherche à produire la première cible. C'est pour cela qu'il est conseillé de mettre une cible fictive par défaut en premier (mais pas la cible `clean`).

Si l'on souhaite exécuter `make` sur un fichier de configuration particulier, on utilise l'option `-f` :

> `make -f monmakefile`

Si l'on souhaite exécuter `make` en définissant une nouvelle variable, ou en modifiant le contenu d'une variable existante, on fait la définition en argument de la commande `make` :

> `make OBJETS='toto.o'`

En effet, cette variable sera alors déjà définie avant la lecture du fichier de configuration, et la simple ligne

```
OBJETS = complexe.o complexe-main.o
```

ne modifie pas une variable déjà définie. Pour définir à nouveau une variable déjà définie, il faut le préciser avec la commande `override`.

Enfin, pour demander l'exécution (si besoin) de la commande associée à une cible précise, on passe cette cible sur la ligne de commande. Ainsi, le `Makefile` classique pour un projet C est de la forme présentée dans le programme 26.

Makefile3

```
GCC = gcc -O
OBJETS = complexe.o complexe-main.o

do : complexe

%.o : %.c
        $(GCC) -c $<

complexe : $(OBJETS)
        $(GCC) $(OBJETS) -o complexe

clean :
        rm -f $(OBJETS) complexe core *~

complexe.o : complexe.c complexe.h
complexe-main.o : complexe-main.c complexe.h
```

Programme 26: Makefile : classique C (`Makefile3`)

Son utilisation est alors

> `make -f Makefile3` pour créer l'exécutable et

> `make -f Makefile3 clean` pour faire " le ménage ".

10.2 Une gestion des versions : `cvs`

Lors de gros projets, on aime garder les versions stables successives, afin de revenir en arrière en cas de découverte de bogues. On aime aussi avoir un historique des différentes étapes. Puis enfin, si on travaille à plusieurs sur ce projet, les mises à jours ne sont pas aisées.

L'utilitaire `cvs` (Concurrent Versions Control) fait tout cela pour vous !

Il y a donc une base, un répertoire accessible par tous les utilisateurs du projet, éventuellement sur une machine distante, dans laquelle seront stockées les différentes versions. Ce répertoire est défini par la variable système `CVSROOT` :

> `setenv CVSROOT=$HOME/cvsroot`

ce répertoire est alors créé et initialisé par la commande

> `cvs init`

10.2.1 Création d'un projet

10.2.1.1 En partant de rien !

Nous voulons commencer un nouveau projet, avec une gestion des versions. Pour cela, il faut créer le(s) répertoire(s) chez soi

> `mkdir newproject`

puis, alors, créer le répertoire correspondant dans la base centrale

> `cd newproject`
> `cvs import -m "Création du répertoire" newproject projet start`

10.2.1.2 En partant d'un projet commencé

Nous voulons gérer les versions d'un projet déjà commencé. Pour cela, il faut insérer tous les fichiers existants dans la base centrale. Une fois dans le répertoire contenant le projet en cours,

> `cvs import -m "Insertion des fichiers" newproject projet start`

10.2.2 Fichier d'information

Pour créer le répertoire CVS dans le répertoire du projet en cours, taper

> `cd ..`
> `cvs checkout newproject`

Maintenant, les ajouts, modifications, suppressions de fichiers et gestion de versions peuvent commencer.

10.2.3 Modifications de la structure

10.2.3.1 Ajout de fichier/Répertoire

Pour ajouter un élément (fichier ou répertoire) à la liste de ceux gérés par `cvs`, par exemple `Makefile`, on tape

> `cvs add Makefile`

Cet ajout sera effectif lors de la prochaine mise à jour de la base.

10.2.3.2 Suppression de fichier/répertoire

Pour supprimer un élément (fichier ou répertoire) à la liste de ceux gérés par `cvs`, par exemple `Makefile.old`, on tape
> `cvs remove Makefile.old`
Cette suppression sera effective lors de la prochaine mise à jour de la base.

10.2.3.3 Modification effective

Les modifications ci-dessus, puis les modifications des contenus des fichiers sont locales. Pour les rendre effectives dans la base, et donc accessibles à tous, il faut faire
> `cvs commit`
Cette opération modifiera la base.

Remarque : Il faut remarquer que quelqu'un peut avoir modifié un fichier sur lequel vous avez travaillé. Ainsi, cette opération peut rencontrer des conflits. Cependant, `cvs` tente de repérer les lignes modifiées par chaque utilisateur, et de faire les mises à jour au mieux. Si `cvs` ne sait pas gérer le conflit rencontré, il vous demande de l'aide. Pour éviter au maximum ces conflits, il suffit de souvent synchroniser les répertoires (personnel et base de `cvs`) avec `commit` pour mettre à jour la base de `cvs`, et `update` pour mettre à jour son répertoire personnel (voir ci-dessous).

10.2.4 Intégration des modifications

Comme précisé ci-dessus, ce gestionnaire de versions permet le suivi de projet par plusieurs utilisateurs. Il faut donc mettre à jours son propre répertoire en fonction des mises jour de la base par les autres programmeurs (pour éviter au maximum les conflits). Pour cela, avant de se mettre au travail, on met à jour sa base de travail personnelle
> `cvs update`

10.2.5 Gestion des versions

Cet utilitaire permet de gérer les versions. Par défaut, les fichiers commencent avec un numéro de version 1.1. Lorsque le projet a atteint un état satisfaisant, on le fige dans un nouveau numéro de version
> `cvs commit -r2.0`
Ainsi, l'état actuel est répertorié dans le numéro de version 2.0.

À tout moment, il est possible de rétablir un version ancienne :
> `cvs checkout -r1.1`
crée le répertoire avec les fichiers dans l'état d'origine (version 1.1), où on le souhaite, après avoir supprimé notre propre version par exemple.

10.2.6 Commentaires

Lors de la plupart des opérations, des commentaires peuvent être insérés. Ils apparaîtront dans les " log " :
> `cvs log Makefile`

Soit ces commentaires sont mis sur la ligne de commande, par l'intermédiaire du paramètre `-m`, comme présenté dans les exemples de `cvs import` ci-dessus, soit un éditeur de texte est lancé afin que vous saisissiez ces commentaires avant d'achever la mise à jour.

10.3 Le débogueur : `gdb`

Un utilitaire bien pratique pour résoudre des problèmes de programmation apparemment " insolubles ", le débogueur. Il permet de nombreuses opérations, nous allons voir les essentielles. Il faut cependant remarquer qu'une aide en ligne répond à la plupart des questions, en posant la question `help`, suivie éventuellement de la commande sur laquelle on souhaite des détails.

10.3.1 Lancement du débogueur

Pour utiliser le débogueur, il faut compiler le programme avec les symboles de déboguage. Pour cela, on ajoute le paramètre `-g`

> `gcc -g hello.c -o hello`

Ensuite, on le lance sur la ligne de commande, avec

> `gdb hello`

Mais cette utilisation est beaucoup moins conviviale que sous `emacs`. En effet, sous `emacs`, il est possible d'avoir une flèche qui indique l'endroit de l'exécution dans le fichier C. Pour lancer le débogueur sous `emacs`, on tape `Alt-X gdb`. Puis à la question suivante, on saisie le nom de l'exécutable.

10.3.2 Commandes de base

Dans chacune des commandes ci-dessous, il est possible de ne saisir que le début de chaque mot, si aucune ambiguïte ne survient : `r` pour `run`, `b` pour `breakpoints`, etc.

10.3.2.1 `run`

On peut bien entendu lancer l'exécution du programme à déboguer avec `run`. Si le programme ne provoque pas de débordement (erreur fatale), il se termine normalement, sinon il termine avec un

```
Program received signal SIGSEGV, Segmentation fault.
0x80483b3 in main () at segmentation.c:5
```

avec le programme 27 qui accède à de la mémoire non allouée.

10.3.2.2 `where`

La commande `where` précise la position dans la pile d'exécution, ce qui permet de savoir dans quel appel de fonction on se trouve.

segmentation.c

```
#include <stdio.h>
int main ()
{
  int *tab=NULL;
  tab[10]=5;
  return 0;
}
```

Programme 27: Débordement (segmentation.c)

10.3.2.3 up

La commande up fait monter d'un cran dans la pile d'exécution, ce qui permet de savoir où a été fait l'appel de fonction dans laquelle on se trouve.

10.3.2.4 breakpoints

Il est possible de positionner des points d'arrêt afin d'obliger l'arrêt de l'exécution du programme à un endroit précis, en donnant le numéro de la ligne :

```
breakpoints 4
```

10.3.2.5 continue

Cela relance l'exécution jusqu'au prochain point d'arrêt (ou fin de programme).

10.3.2.6 step

Exécute la ligne fléchée, et stoppe l'exécution. On descend alors, si nécessaire, dans la fonction appelée pour stopper l'exécution au début de cette fonction.

10.3.2.7 next

Exécute la ligne fléchée, et stoppe l'exécution sur la ligne suivante dans le programme. En cas d'appel à une fonction, cette fonction est totalement exécutée avant l'arrêt de l'exécution.

10.3.2.8 print

Permet d'afficher ponctuellement le contenu d'une variable

```
print tab
```

affiche le contenu de la variable `tab`, si cette dernière est définie.

10.3.2.9 `display`

Affiche le contenu d'une variable à tout arrêt de l'exécution.

Chapitre 11

Quelques compléments

Sommaire

Dans ce chapitre, nous allons regrouper un certain nombre de renseignements utiles, telle que la manipulation des chaînes de caractères et des fichiers. Leurs descriptions plus complètes pourront être trouvées avec un appel au `man` (ex : `man strlen`).

11.1 Les chaînes de caractères

11.1.1 Structure d'une chaîne de caractères

Une chaîne de caractères est en fait un tableau de caractères. La fin de la chaîne est désignée par le caractère `'\0'`. Il s'agit donc aussi d'un pointeur sur un caractère (`char *`). Mais des fonctions particulières permettent de manipuler ces tableaux de caractères plus simplement que des tableaux quelconques. Le programme 28 en regroupe certaines.

11.1.2 Quelques commandes sur les chaînes de caractères

Des commandes permettent de déterminer la longueur d'une chaîne, de comparer deux chaînes, de copier le contenu de l'une dans l'autre ou de concaténer deux chaînes :

palindrome.c

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

int palindrome(char *s)
{
  char s1[80];
  int i,l = strlen(s);
  for (i=0; i<l; i++)
    s1[i] = s[l-i-1];
  s1[l] = '\0';
  return (!strcmp(s,s1));
}

int main (int argc, char *argv[])
{
  char s[80];
  int t;
  strcpy (s,argv[1]);
  t = palindrome(s);
  if (t) printf("%s est un palindrome\n",s);
  else printf("%s n'est pas un palindrome\n",s);
  return 0;
}
```

Programme 28: Palindrome (`palindrome.c`)

- `strlen`, qui retourne la longueur de la chaîne (sans le `'\0'` final) ;
- `strcmp` et `strncmp`, qui comparent le contenu de deux chaînes de caractères. La commande `strcmp(s1,s2)` retourne un nombre négative, nul ou positif selon que `s1` est plus petit, égal, ou supérieur à `s2`. La commande `strncmp(s1,s2,n)` fait la même chose, mais en ne considérant que les `n` premiers caractères de `s1` ;
- `strcpy` et `strncpy`, qui copient le contenu d'une chaîne de caractères dans une autre. La commande `strcpy(s1,s2)` copie la chaîne `s2` dans la chaîne `s1`, à condition que la longueur de `s1` (nombre de cases du tableau de caractères allouées) est suffisante. La commande `strncpy(s1,s2,n)` fait la même chose, mais en ne copiant que les `n` premiers caractères de `s2` ;
- `strcat` et `strncat`, qui fonctionne comme les deux fonctions précédentes, à la différence que la chaîne `s2` est concaténée à la suite de la chaîne `s1` (si la place nécessaire est disponible).

Remarque : pour ces quatre dernières commandes `strcpy` et `strncpy` puis `strcat` et `strncat`, la chaîne réceptrice doit être allouée, de taille suffisante, sous peine, soit d'une

erreur de segmentation dans le meilleur des cas, soit une exécution erronée et aléatoire du programme.

Remarque : Il est important de retenir que les deux déclarations suivantes :

```
char v[] = "abcd";
char *p = "abcd";
```

ne sont pas équivalentes. La première déclare un vecteur tableau de 5 caractères, initialisé avec les caractères 'a', 'b', 'c', 'd', '
0'; la seconde déclare un pointeur de caractère initialisé avec l'adresse de la constante chaîne de caractères "abcd". La différence fondamentale entre ces deux objets est que le contenu du tableau est modifiable alors que celui de la chaîne ne l'est pas. Généralement, les constantes littérales chaînes de caractères sont rangées dans des blocs de mémoire protégés physiquement en écriture. Toute tentative de modification provoque une erreur. Par exemple provoque une erreur lors de la tentative de modification de *p.

tblvsptr.c

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

char v[] = "abcd";
char *p = "abcd";

int main(){

  printf("Modification de v[0]\n");
  v[0]='A';
  printf("OK\nModification de *p\n");
  *p='A';
  printf("OK\n");

  return EXIT_SUCCESS;
}
```

Programme 29: Tableau vs. Pointeur (tblvsptr.c)

Remarque : Il faut également faire attention à l'erreur suivante. Un identificateur de tableau lettres est utilisé dans la partie gauche d'une affectation. Il illustre le fait qu'un identificateur de tableau est une constante et ne peut faire l'objet d'une affectation. En C, un tableau n'est pas une variable, mais une suite de variables. Il n'est pas directement modifiable au moyen d'une affectation. En revanche, il est possible de modifier chacun de ses éléments.

tblvsptr1.c

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

int main(){

  char majuscules[] = "ABCDEFGHIJKLMNOPQRSTUVXYZ";
  char lettres[26];

  lettres = majuscules;
  return EXIT_SUCCESS;
}
```

Programme 30: Tableau vs. Pointeur (`tblvsptr1.c`)

11.2 Gestion des fichiers

Nous avons vu comment passer des arguments sur la ligne de commande (dans les arguments de la fonction `main`) ou comment prendre des valeurs tapées au clavier pendant l'exécution du programme, avec la commande `scanf`. Mais parfois, les données proviennent d'un fichier, ou des résultats doivent être stockés dans un fichier. Cette section présente les rudiments de la gestion des fichiers en C.

11.2.1 Type fichier

Tout d'abord, un fichier est identifié par un objet de type `FILE *`, il s'agit d'un *flux*. Trois flux sont définis par défaut :
- `stdin`, l'entrée standard, *i.e.* le clavier
- `stdout`, la sortie standard, *i.e.* l'écran
- `stderr`, la sortie des erreurs standard, *i.e.* l'écran

11.2.2 Création d'un fichier

Pour créer un nouveau flux, il faut l'associer à un fichier physique, désigné sous UNIX par une chaîne de caractères (le chemin relatif ou absolu). Ce lien s'effectue par la commande `fopen` :

```
FILE * fopen (const char * path, const char * mode)
```

où `path` est la chaîne de caractères qui désigne l'emplacement du fichier dans la hiérarchie du disque, et `mode` désigne le mode de l'ouverture :

- `r` (read), le fichier est ouvert en lecture seule. Le *curseur* se positionne en tête de fichier.
- `w` (write), le fichier est créé (ou recréé) vide en écriture. Le *curseur* se positionne en tête de fichier.
- `a` (append), le fichier est ouvert en écriture. S'il n'existe pas, il est créé vide. Le *curseur* se positionne en fin de fichier.

11.2.3 Lecture/Écriture

Il est possible d'utiliser `fprintf` et `fscanf`, qui fonctionnent exactement comme `printf` et `scanf` (voir chapitre 4) avec un premier argument supplémentaire : le fichier.

Pour lire ou écrire des caractères, il est possible d'utiliser les fonctions `fgetc` et `fputc` (voir programme 31).

Après chaque lecture, le *curseur* avance dans le fichier lu. Il en est de même pour la position d'écriture.

11.2.4 Fin de fichier

À tout moment, on peut savoir si l'on est arrivé à la fin du fichier avec la commande `feof`. Étant donné un flux, cette commande retourne `0` si la fin n'est pas atteinte, et un nombre non nul dans le cas contraire.

```
int feof(FILE * file)
```

11.2.5 Clôture

Lorsque le fichier n'est plus utile, on le ferme pour libérer un flux (le nombre de flux étant limité) :

```
int fclose(FILE * file)
```

11.3 C 99

La norme C99 représente une évolution importante du langage. On peut citer les changements suivants :

- la possibilité de mélanger les déclarations et les instructions,
- l'introduction du type booléen et complexe,
- diverses extensions venues enrichir les mécanismes de déclaration, comme les tableaux de taille variable ou les fonctions en ligne,
- l'ajout de nouveaux types entiers.

Les modifications les plus importantes concernent la bibliothèque de fonctions standard dont le nombre d'environ 150 dans la première version est passé à plus de 480.

11.4 La bibliothèque standard

Le langage C "pur" se limite aux déclarations, expressions, instructions et blocs d'instructions, structures de contrôle, et fonctions. Il n'y a pas d'instructions d'entrée/sortie par exemple. Celles-ci sont implémentées sous la forme de fonctions contenues dans la bibliothèque du langage C. Cette bibliothèque, initialement conçue pour interagir avec le système UNIX était à l'origine très dépendante de l'implémentation, et cela a constitué pendant plusieurs années la difficulté de portage des programmes C. Ce problème a été résolu avec, la norme ISO/IEC 9899, qui fixe et décrit l'ensemble des fonctions standard du langage, et d'autre part avec la norme POSIX.1 qui précise l'interface avec le système d'exploitation. Un programme se limitant aux fonctions standard est portable dans tout l'environnement C standard.

Des erreurs peuvent subvenir dans les appels de fonctions de la bibliothèque standard. Chaque erreur est codifiée par un numéro associé à un message spécifique. Le numéro de l'erreur est accessible au moyen du symbole externe `errno` assimilable à un identificateur de variable. Il reçoit un code d'erreur lors d'un déroulement incorrect de certaines fonctions de bibliothèque. La valeur de `errno` n'est pas automatiquement remise à zéro par les fonctions de la bibliothèque. *Il est possible d'identifier une erreur survenue lors de l'exécution de certaines des fonctions de la bibliothèque standard en testant la valeur de la variable globale* `errno`*. Il est dans ce cas indispensable de mettre* `errno` *à zéro avant d'effectuer l'appel à la fonction.*

Voici un petit exemple affichant les valeurs des erreurs. La fonction `char *strerror(int numero-d-erreur)` retourne le message d'erreur, alors que la fonction `void perror(const char *en-tete)` affiche la concaténation de la chaîne `en-tete`, de la chaîne " :" et du message d'erreur.

copie.c

```
#include <stdio.h>

int TestFinDeFichier(FILE * fichier) {
  return (feof(fichier));}

void EcritCaractere(FILE * out, char val) {
  fputc(val,out);}

char LitCaractere(FILE * in) {
  return (fgetc(in));}

int main (int argc, char *argv[])
{ FILE *in,*out;
  char fichierin[80], fichierout[80];
  char c;

  if (argc!=3) {
    printf("Usage: copie <in> <out> \n"); exit(0);}

  strcpy(fichierin,argv[1]);
  strcpy(fichierout,argv[2]);

  in=fopen(fichierin,"r");
  if (!in) {
    printf("Fichier '%s' inexistant \n",fichierin); exit(0);}
  out=fopen(fichierout,"w");
  if (!out) {
    printf("Problème de création du fichier '%s' \n",fichierout);
    exit(0);}

  c=LitCaractere(in);
  while(!TestFinDeFichier(in)) {
    EcritCaractere(out,c);
    c=LitCaractere(in);}
  fclose(in);
  fclose(out);
  return(0);
}
```

Programme 31: Gestion des fichiers (`copie.c`)

perror.c

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <errno.h>
#include <locale.h>
#include <math.h>

#define PERROR(appel) errno = 0; appel; print_error("- " #appel)

static void print_error(char *source) {
  int e = errno;

  perror(source);
  printf("errno: %d -- %s\n\n", e, strerror(e));
}

int main() {

  PERROR(acos(2.0));
  PERROR(fopen("/", "w"));
  PERROR(fopen("?", "r"));
  PERROR(strtol("1234567890", 0, 0));
  PERROR(strtol("12345678900", 0, 0));
  PERROR(malloc(100000000000));
  PERROR(malloc(1000000000000));

  return EXIT_SUCCESS;
}
```

Programme 32: Erreurs dans la bibliothèque standard (`perror.c`)