PREMIER MINISTRE
Secrétariat général de la défense nationale
Direction centrale de la sécurité des systèmes d'information
Centre de formation en sécurité des systèmes d'information
Brevet d'études supérieures en sécurité des systèmes d'information (BESSSI)

LE LANGAGE C UTILISÉ SOUS LINUX

par

Florent Chabaud

Secrétariat général de la défense nationale
Direction centrale de la sécurité des systèmes d'information
SDS/LTI
51, boulevard de Latour-Maubourg

75700 Paris 07 SP

Version 1.6

Table des matières

Table des matières 3
1 Introduction 7
1.1 Avertissement 7
1.2 Gestion de configuration 7
1.3 Suivi des modifications 7
2 Présentation du langage C 9
2.1 Syntaxe élémentaire 9
2.1.1 Grammaire 9
2.1.1.1 Déclarations 9
2.1.1.2 Initialisations 10
2.1.2 Fonctions de base 11
2.1.2.1 Fonctions arithmétiques 11
2.1.2.2 Branchement conditionnel 12
2.1.2.3 Branchement conditionnel multiple 13
2.1.2.4 Boucle conditionnelle 14
2.1.3 Fonctions 15
2.1.4 Domaines d'existence des variables 15
2.1.5 Directives de compilation 16
2.1.5.1 Préprocesseur 16
2.1.5.2 Inclusions 16
2.1.5.3 Macros 16
2.1.5.4 Options de compilation 17
2.2 Affichage de données 18
2.3 Compilation 19
3 Où le déverminage devient nécessaire 23
3.1 Tableaux et chaînes de caractères 23
3.2 Allocation dynamique de mémoire 23
3.2.1 Qu'est-ce qu'un pointeur ? 23
3.2.2 Allocation dynamique de mémoire 24
3.2.3 Déréférenciation 26
3.2.4 Équivalence tableau-pointeur 26
3.2.5 Pointeur de fonction 28
3.3 Gestion des options 28
3.4 Gestion des fichiers 28
3.4.1 Ouverture de fichier 29
3.4.2 Écriture dans un fichier 29
3.4.3 Lecture d'un fichier 30
3.4.4 Autre fonctions 30

3.4.4.1 Accès direct au fichier 30
3.4.4.2 Lecture de fichier texte 31
3.4.4.3 Fin de fichier 31
3.5 Déverminage par **gdb** 32
3.5.1 Intérêt du déverminage 32
3.5.2 Compilation en mode déverminage 32
3.5.3 Lancement de l'interface graphique de déverminage . . 32
3.5.4 Fonctions principales de **gdb** 34
3.5.5 Règles de programmation pour faciliter le déverminage 37
4 Structures de données évoluées 41
4.1 Liste chaînées 41
4.1.1 Listes, piles, files, etc. 41
4.1.2 Manipulation des structures de données 42
4.2 Récursion et algorithmes de tri 43
4.2.1 Récursion 43
4.2.2 Algorithmes de tri 43
4.2.2.1 Tri par insertion 43
4.2.2.2 Tri par fusion 45
4.3 Arbres binaires 45
4.3.1 Structure de données 45
4.3.2 Complexité de la recherche dans un arbre binaire ordonné 45
4.3.3 Équilibrage d'un arbre binaire 47
4.4 Tables de hachage 47
4.5 Bibliothèques et compilation partagée 48
4.5.1 Compilation partagée 48
4.5.1.1 Principe 48
4.5.1.2 Make 49
4.5.1.3 Règles de programmation partagée 50
4.5.2 Fabrication d'une bibliothèque 51

Annexes **53**
A Exemples de programmes 55
B Exercices de programmation en C 67
B.1 Syntaxe élémentaire 67
B.2 Allocation dynamique de mémoire 67
B.3 Structures de données 67
C Corrections commentées 69
D Commandes de base **Unix** 91
E Index 99

Table des figures

Code 1 : Un programme élémentaire 18
Code 2 : Passage par adresse 24
1 Tableau à deux entrées et double pointeur 27
2 Lancement de **gdb** sous **emacs** 33
3 Lancement de **gdb** sous **emacs** : saisie du fichier exécutable . . . 34
4 Interface de **gdb** sous **emacs** 38
5 Point d'arrêt **gdb** sous **emacs** 39
6 Liste chaînée de trois éléments 41
7 Insertion d'éléments dans une liste chaînée 42
8 Insertion d'éléments dans un arbre binaire de recherche 46
9 Équilibrage d'un arbre binaire 48
Code 3 : Lecture d'un fichier par `fread` 55
Code 4 : Création et affichage d'une file 58
Code 5 : Équilibrage d'un arbre binaire de recherche 62
Code 6 : Exemple de fichier prototype 65
Code 7 : Lecture d'un fichier ligne à ligne par `fgets` 69
Code 8 : Tri par insertion et par fusion 76

1 Introduction

1.1 Avertissement

Ce cours se veut une simple introduction au langage de programmation CṠon objectif est de fournir aux étudiants du BESSSI un niveau minimal de programmation dans ce langage leur permettant d'aborder la suite du cursus avec les bases nécessaires. En effet, les performances d'exécution de ce langage de programmation en font un passage obligé pour tout ce qui concerne la cryptographie.

Comme souvent en informatique, la maîtrise de ces outils ne s'obtient que par une pratique régulière, et ce document a pour seule vocation de guider les premiers pas pour les rendre moins fastidieux.

Le cours est prévu sur dix demi-journées. Les cinq premières alterneront cours magistraux et exercices pratiques de mise à niveau. Les cinq suivantes auront pour objectif la réalisation en langage C d'une bibliothèque de manipulation de structures de données évoluées : listes chaînées, arbres binaires, tables de hachage.

1.2 Gestion de configuration

Version	: 1.6
Date	: 02/10/2003
Module CVS	: UnixC
Fichiers	: UnixC.tex Makefile sedtags.sh extract.awk version.txt cours.sty bib.tex exos.tex cmdunix.tex index.tex arbres.tex bibliotheque.tex compilation.tex fichiers.tex gdb.tex hachage.tex listes.tex options.tex pointeurs.tex printf.tex syntaxe.tex tableaux.tex tri.tex pointeur.fig liste.fig insertionliste.fig insertionarbre.fig equilibre.fig jpg_emacs_gdb.fig jpg_emacs_gdb2.fig jpg_emacs_gdb3.fig jpg_emacs_gdb4.fig hello pointeur fwrite tri fgets liste prototype arbres
Hash MD5	: fa0d94631f23575c19c6e8ed256e40dd

1.3 Suivi des modifications

1.6	02/10/2003	Version envoyée au CFSSI pour impression
1.5	02/10/2003	Version modifiée suite aux remarques des élèves
1.4	12/09/2003	Version modifiée suite à la réorganisation pédagogique du BESSSI
1.3	27/08/2003	Réactivation de l'index et du suivi des dates
1.2	27/08/2003	Transformation du format pour le suivi de version
1.1	27/08/2003	Version initiale correspondant au cours distribué en 2002-2003

2 Présentation du langage C

2.1 Syntaxe élémentaire

2.1.1 Grammaire

La grammaire du langage C est relativement simple ; un **bloc** consiste en :

1. Déclarations.
2. Fonctions.

Un programme en C est lui-même un **bloc**.

2.1.1.1 Déclarations

Tous les objets utilisés dans un programme C doivent être déclarés avec un type défini. Les principaux types possibles sont :

int désigne le type entier[1], qui correspond le plus souvent à un entier codé sur 32 bits, donc compris entre -2^{31} et $2^{31}-1$.

char désigne un type entier codé sur 8 bits, donc compris entre -128 et 127.

double désigne un nombre en virgule flottante[2]

Comme toute commande en C se termine par un point-virgule, pour déclarer une variable i de type entier, on écrira donc :

```
int i;
```

Il est possible de combiner ces types dans des structures contenant plusieurs champs, par exemple :

```
struct nombre_complexe {
  double x,y;
} z;
```

déclare z comme une variable contenant deux nombres en virgule flottante. Ces deux nombres sont alors obtenus par la syntaxe z.x et z.y.

Mais il est aussi possible de déclarer de nouveaux types grâce à l'opérateur typedef ; en voici quelques exemples :

– typedef int Entier ; permet d'utiliser `Entier` à la place de `int`.

[1] Il existe d'autres types d'entier, mais malheureusement ces types ont des tailles variables selon les machines :
short désigne normalement un entier codé sur 16 bits.
long désigne un entier codé sur un mot machine, donc actuellement 32 ou 64 bits.
Sur certaines machines anciennes à microprocesseur 16 bits, on peut trouver des tailles différentes.

[2] Il existe aussi un type float définissant un nombre en virgule flottante de simple précision. La totalité des fonctions mathématiques étant définies sur la base de nombres en double précision, il est préférable de ne pas utiliser le type float.

– typedef enum {Dimanche,Lundi,Mardi,Mercredi,Jeudi,Vendredi,Samedi} Jour ; permet de déclarer un type d'entier nouveau à 7 valeurs. Cette syntaxe code par défaut :
  typedef enum {Dimanche=0,Lundi=1,Mardi=2,...,Samedi=6} Jour ;
  mais rien n'empêche de modifier ces valeurs tant que des entiers sont utilisés.
– typedef struct nombre_complexe { double x,y } Complexe ; définit le nouveau type Complexe.

Les nouveaux types peuvent alors s'utiliser comme les anciens. Par exemple Complexe z ; déclare la variable z comme un nombre complexe.

Note 1 *Le mot clé* unsigned *permet de déclarer des nombres non signés. Ainsi, un caractère alphanumérique codé en ASCII (de 0 à 255) est de type* unsigned char. *Son codage peut être effectué grâce à la syntaxe* 'c'*, où les apostrophes délimitent le caractère. On notera aussi les syntaxes particulières suivantes pour quelques caractères spéciaux :*
  '\n' *code pour le retour chariot (*newline*) ;*
  '\t' *code pour la tabulation ;*
  '\'' *code pour l'apostrophe ;*
  '\" ' *code pour le guillemet ;*
  '\\' *code pour le backslash (\).*

Note 2 *Le* C *permet de convertir des valeurs d'un type à un autre. Il s'agit bien d'une conversion sur les valeurs, et non d'une conversion sur les variables, qui elles, conservent toujours le type de leur déclaration. Cette conversion s'effectue par un opérateur "*cast*". La syntaxe de cet opérateur est un peu particulière. Le nouveau type est placé entre parenthèses devant la valeur à convertir. Ainsi*

```
a = (int) 2.3;
```

affecte à la variable a *la valeur*[3] *entière* 2. *Il existe des règles de conversion implicite, mais il est prudent en* C *de toujours supposer qu'aucune conversion n'est effectuée par défaut. L'un des défauts (mais qui dans certains cas est aussi un avantage) du langage* C *est qu'il est toujours possible d'effectuer la conversion d'un type en un autre type même lorsqu'une telle conversion n'a pas de sens (voir note 8, page 25).*

2.1.1.2 Initialisations

En C la déclaration n'effectue **aucune** initialisation. Supposer qu'une variable déclarée est initialisée à zéro est, par exemple, une erreur.

[3] Il ne s'agit pas de la partie entière mathématique, car a=(int)-1.7 donne la valeur -1 et non la valeur "mathématique" -2.

2.1.2 Fonctions de base

2.1.2.1 Fonctions arithmétiques

Les fonctions arithmétiques de base sont disponibles :

= L'affectation à une valeur : x = 2 ;.

+ Addition.

- Soustraction.

* Multiplication. Dans le cas de la multiplication d'un entier par un flottant, le résultat est un flottant.

/ Division. Dans le cas entier, il s'agit de la division euclidienne. On obtient donc un entier. Si l'un des nombres est flottant, le résultat est flottant.

% Réduction modulaire. Dans le cas de deux entiers uniquement, `x%y` fournit le reste de la division euclidienne `x/y`.

& Effectue un "et" bit-à-bit de deux entiers. Par exemple, (12 & 22) vaut 4, car :

12	se code en binaire	00001100
22	se code en binaire	00010110
12&22	donne donc en binaire	00000100

| Effectue un "ou" bit-à-bit de deux entiers. Par exemple, (12 | 22) vaut 30, car :

12	se code en binaire	00001100
22	se code en binaire	00010110
12|22	donne donc en binaire	00011110

ˆ Effectue un "ou-exclusif" bit-à-bit de deux entiers. Par exemple, (12 ˆ 22) vaut 26, car :

12	se code en binaire	00001100
22	se code en binaire	00010110
12ˆ22	donne donc en binaire	00011010

˜ Effectue une négation bit-à-bit d'un entier. Par exemple `˜0` donne l'entier dont tous les bits sont à un.

Note 3 *Le* C *est un langage qui dispose de certaines astuces performantes. Des opérateurs permettent ainsi des raccourcis pratiques :*

- i += 2 *effectue* i = i + 2. *Les opérateurs* -=, *=, /=, %=, &=, |= *et* ˆ= *effectuent des opérations analogues.*
- ++i *effectue* i = i + 1. *L'opérateur* - - *existe aussi.*

– i++ *effectue* i = i + 1*, mais a pour valeur la valeur initiale. Donc*

```
i = 0;
x = (++i);
y = (i--);
```

donne à i *la valeur 0, à* x *la valeur 1 et à* y *la valeur 1.*
Combiner plusieurs de ces opérateurs dans une même opération peut s'avérer risqué.

2.1.2.2 Branchement conditionnel

La syntaxe du branchement conditionnel est la suivante :

`if (condition) {`

bloc1

`} else if (condition) {`

bloc2

`} else {`

bloc3

`}`

Ici encore on remarque que les **bloc**s sont encadrés par des accolades. Si le **bloc** est limité à une seule instruction sans déclaration, alors les accolades peuvent être omises. Bien entendu, les "else if" et "else" et les **bloc**s correspondants sont facultatifs.

La condition testée peut être n'importe quelle opération arithmétique, ou d'affectation. Elle est vraie si le résultat est non nul, fausse sinon (par exemple `x=1` est vrai). Des opérateurs de comparaison entre nombres existent aussi :

== Teste l'égalité. C'est une erreur très classique d'écrire if (x = 1) en pensant if (x == 1), et c'est une erreur difficile à retrouver, donc le mieux est d'essayer d'y penser[4].

!= Teste la différence.

>= On peut aussi comparer en utilisant ">=", ">", "<=" ou "<".

Deux conditions peuvent être combinées par des opérateurs booléens :

&& correspond à l'opérateur booléen ET.

[4]On note toutefois que les compilateurs modernes émettent parfois un avertissement pour ce genre de syntaxe et qu'on peut dans ce cas forcer la compilation en écrivant if ((x=1)) si on souhaite réellement regarder le résultat d'une affectation.

|| correspond à l'opérateur booléen OU.

! Effectue la négation booléenne d'une condition. Par exemple :

```
!((x >= 2) && (x <= 4))
```

correspond à

```
((x < 2) || (x > 4))
```

Note 4 *Encore, une fois, le* C *est un langage qui permet certaines astuces performantes. Ainsi, lors de la combinaison booléenne de conditions, il peut se faire qu'une condition ne soit pas évaluée, par exemple :*

```
if ((x == 0.0)||(1.0/x == 1.0))
  x=1.0;
```

est valable et ne donne pas d'erreur à l'exécution car si `x` *est nul, la condition est forcément vraie, et* `1.0/x` *n'a pas besoin d'être calculé.*

2.1.2.3 Branchement conditionnel multiple

Le branchement conditionnel multiple est possible en C selon une valeur numérique :

```
switch(valeur)
{
  case val1:
    {
```

bloc1

```
    }
    break;
  case val2:
    {
```

bloc2

```
    }
    break;
  default:
    {
```

bloc par défaut

```
    }
    break;
}
```

2.1.2.4 Boucle conditionnelle

Il est possible d'effectuer en C une boucle conditionnelle

```
while(condition)
  {
```

bloc

```
  }
```

qui s'exécute tant que la condition est vraie (c'est-à-dire non nulle). Une deuxième syntaxe est possible, qui assure que la boucle est exécutée au moins une fois :

```
do {
```

bloc

```
  } while(condition);
```

Enfin la syntaxe :

```
for(initialisation ; condition ; opération)
  {
```

bloc

```
  }
```

est équivalente à la syntaxe :

```
initialisation ;
while (condition)
  {
```

bloc

```
  opération ;
  }
```

Elle permet en particulier d'effectuer des boucles numérotées facilement :

for(i=2 ;i<=100 ;i++)

permet par exemple, d'effectuer une boucle avec i variant de 2 à 100 inclus[5]

[5] Les séquences d'initialisation, de condition et d'opération d'une boucle for peuvent comporter plusieurs commandes séparées par des virgules. On peut par exemple écrire :

```
for(i=2, j=0 ; i<=100 && j>0 ; i++, j=fct(i)) {
        bloc
}
```

2.1.3 Fonctions

Une fonction se déclare aussi, avec le type qu'elle retourne[6] et les arguments qu'elle utilise. Par exemple :

```
double Multiplication(int n, double x);
```

Il est nécessaire de répéter le type de chaque paramètre. On aura donc :

```
double Mult(int n, double x, double y);
```

Si une fonction n'a pas de paramètre on l'indique par le type void.

```
double Init(void);
```

La syntaxe de définition d'une fonction reprend sa déclaration suivie ensuite du **bloc** qui la définit, par exemple

```
double Multiplication(int n, double x)
{
  double ret;

  ret = n*x;
  return(ret);
}
```

On note l'emploi des accolades qui délimite un **bloc**. Seul le **bloc** général d'un programme peut se passer de ces accolades.

Le mot clé `return` permet de terminer la fonction et de retourner la valeur indiquée[7].

Pour être exécutable, un programme doit comporter la fonction spéciale `main`, et c'est le **bloc** qui la constitue qui sera exécuté.

2.1.4 Domaines d'existence des variables

1. Toutes les variables déclarées dans un **bloc** sont locales à ce **bloc**.
2. Tout **bloc** inclus dans un autre hérite de ses variables. Redéclarer celles-ci est donc inutile, voire dangereux car les modifications apportées aux variables dans le **bloc** inclus ne seraient pas prises en compte dans le **bloc** principal (voir règle 1).

[6] Dans le cas d'une fonction qui n'a aucune valeur à retourner, le type à utiliser est `void`.

[7] Dans le cas d'une fonction de type `void`, on indique simplement `return ;`.

3. Lors d'un appel de fonction, les variables et les paramètres de la fonction sont locaux. Ils prennent à chaque nouvel appel une valeur, aléatoire pour les variables, fixée par l'appel pour les paramètres[8].

2.1.5 Directives de compilation

2.1.5.1 Préprocesseur

Avant la compilation proprement dite d'un programme C, une première étape effectue une première transformation du fichier source en fonction des directives de compilation qui y apparaissent. Une directive de compilation est indiquée par un caractère "#" situé en début de ligne suivie de la syntaxe de la directive.

2.1.5.2 Inclusions

Il est possible ainsi d'inclure un fichier par la directive #include <fichier>. Ceci est par exemple utile lorsque plusieurs déclarations et/ou définitions de types sont utilisées dans des fichiers sources séparés. Il est traditionnel d'indiquer un fichier contenant des déclarations, des directives de compilation, des définitions de type par le suffixe ".h" (pour *Headers*), tandis que les fichiers sources proprement dits sont indiqués par le suffixe ".c". Une inclusion correspond exactement à la recopie du fichier inclus à l'emplacement du #include dans le fichier. Les directives de compilation qui apparaissent dans le fichier inclus sont donc appliquées dans la suite du fichier appelant. La plupart des fichiers C commencent par #include <stdlib.h> qui inclut des déclarations très utiles.

2.1.5.3 Macros

Bien que ce qui suit ne soit pas indispensable à comprendre, les possibilités offertes par les macros quant à la souplesse de la programmation les rendent très utiles. La syntaxe d'une macro utilise le mot clé "#define" :

```
#define MACRO(VARA,VARB,VARC) DÉFINITION
```

Une macro est une directive qui permet de remplacer dans un fichier toutes les occurrences ultérieures de MACRO(x,y,z) par DÉFINITION, dans laquelle les valeurs de VARA, VARB et VARC seront remplacées respectivement par x,y,z. Par exemple, on pourra définir en liaison avec le type Complexe défini section 2.1.1.1, les macros

[8] Pour qu'une variable locale à une fonction (ou à un bloc) conserve sa valeur d'un appel à l'autre, il faut la déclarer statique ; par exemple static int x ;. La valeur de x à la sortie de la fonction sera ainsi conservée lors du prochain appel à la fonction.

```
#define Re(Z) ((Z).x)
#define Im(Z) ((Z).y)
```

de sorte qu'un Complexe z; sera décrit par la suite par Re(z) et Im(z). Les avantages de cette syntaxe sont les suivants :

- la lisibilité du programme est accrue;
- pour un accès à une structure, ce qui est le cas présent, le fait d'utiliser des macros dans un programme permet ultérieurement de modifier la structure sans avoir à changer tout son programme : seules les macros sont à modifier en conséquence;
- contrairement à une fonction, une macro peut être utilisée dans la partie gauche d'une instruction d'affectation : Re(z)=2.0 est correct;
- une macro étant remplacée dans le source du programme, il n'y a pas d'appel de fonction, ce qui peut être beaucoup plus rapide pour une opération courte.

Note 5 *Cependant, l'emploi des macros peut s'avérer périlleux dans certains cas. Il faut en particulier veiller, dans la* DÉFINITION *à toujours entourer les arguments d'une macro par des parenthèses. D'autre part, lors de l'utilisation d'une macro, il ne faut pas utiliser d'opérateurs du type "++", car si l'argument de la macro apparaît plusieurs fois dans la définition, l'opérateur sera appliqué plusieurs fois.*

Note 6 *Il est possible de limiter la définition d'une macro à une portion de programme en utilisant* #undef MACRO.

2.1.5.4 Options de compilation

Les directives de compilation permettent aussi de ne compiler une portion de source du programme que sous condition. Considérons l'exemple suivant :

```
#if !defined(NIVEAU)
# define NIVEAU 2
#endif

#if NIVEAU > 0
```

bloc1

```
# if NIVEAU > 1
```

bloc2

```
# endif
#else
```

bloc0

```
#endif
```

Si NIVEAU n'a pas été défini antérieurement alors on lui donne la valeur 2. Dans ce cas, les **bloc**s 1 et 2 sont inclus mais pas le bloc 0. La syntaxe des conditions utilisée dans les options de compilation est la même que celle du C. Il est donc possible de combiner des conditions avec les opérateurs présentés au paragraphe 2.1.2.2.

2.2 Affichage de données

Pour afficher à l'écran, la commande à utiliser est printf(format,variables);. Le format est une chaîne de caractères qui décrit la façon d'afficher les variables qui suivent; par exemple printf("x=%d\n",x); affiche x= suivi de la valeur de la variable entière x et d'un retour chariot. La fonction printf est une fonction particulière qui prend un nombre variable d'arguments. Les codes à utiliser dans le format indiquent le type de la variable. Ainsi pour afficher un nombre Complexe, on écrira printf("z=%f+i%f\n",Re(z),Im(z));. Les principaux codes existants pour la chaîne de formatage sont les suivants

Code	Type
%d	Type entier (long, int, short ou char) sorti en base 10.
%u	Type entier non signé (unsigned long, unsigned int, unsigned short ou unsigned char) sorti en base 10.
%f	Type flottant (double ou float) sortie en numérotation décimale.
%e	Type flottant (double ou float) sortie en numérotation décimale avec puissance de 10.
%c	unsigned char sous forme de caractère alphanumérique.
%s	chaîne de caractères.
%%	caractère "%".

Nous avons désormais tous les éléments pour écrire le fameux programme "Hello World" (cf. code 1).

CODE 1 : Un programme élémentaire

```
// Utilisation d'une directive de précompiltaion pour définir une constante
Anglais qui par défaut vaut 1
#ifndef Anglais
# define Anglais 1
#endif
```

// En fonction de cette constante, compiler le programme avec une constante de Message différente.

```
#if Anglais
# define Message "Hello World"
#else
# define Message "Bonjour Monde"
#endif
```

// Les arguments de la fonction principale **main** *seront vus au paragraphe 3.3*

```
int main(void) {
```

// zone de déclaration des variables locales

```
  int i;
```

// zone de fonction

```
  for(i=0;i<=10;i++) {
    printf("%d : en ",i);
    if(Anglais)
      printf("Anglais");
    else
      printf("Français");
    printf(" dans le texte : %s\n",Message);
  }
}
```

2.3 Compilation

Il existe beaucoup de compilateurs C. Sur une machine **Unix** cependant, on n'en trouve généralement qu'un ou deux. La commande cc est généralement réservée au compilateur du constructeur de la machine. On trouve aussi assez souvent le compilateur gcc. Bien que les options de compilation de ces deux programmes soient souvent différentes (malheureusement !), celles qui suivent restent généralement disponibles :

-o permet de préciser le fichier de sortie -o fichier.

-c permet d'effectuer une compilation partagée. Le fichier source ainsi compilé

donne un fichier en .o qui est ensuite directement utilisable pour la compilation du programme final.

-O permet une compilation optimisée. À n'utiliser que lorsque le programme fonctionne de manière correcte.

-g permet d'inclure dans le fichier compilé des informations utiles pour l'utilisation d'un programme de déverminage[9].

-E permet d'obtenir le source du programme une fois effectuées les directives de compilation.

-I permet de préciser les répertoires où chercher les fichiers à inclure. -I. est conseillé pour pouvoir inclure ses propres fichiers .h du répertoire courant.

-D permet de définir une variable de compilation. Par exemple -DDEBUG=2 est équivalent à mettre #define DEBUG 2 au début du programme.

-D permet de définir une variable de compilation. Par exemple -DDEBUG=2 est équivalent à mettre #define DEBUG 2 au début du programme.

-Wall permet d'afficher tous les messages d'anomalies de compilation ne conduisant pas à des erreurs fatales. Ces anomalies peuvent dans la plupart des cas conduire à de véritables erreurs lors de l'exécution du programme, Il est donc fortement conseillé d'utiliser cette option, voire de la compléter avec l'option -Werror qui interdit la compilation dans ce cas.

Ainsi dans l'exemple du tableau 1 avec

```
# gcc hello.c
```

on obtient :

```
# a.out
0 : en Anglais dans le texte : Hello World
1 : en Anglais dans le texte : Hello World
2 : en Anglais dans le texte : Hello World
3 : en Anglais dans le texte : Hello World
4 : en Anglais dans le texte : Hello World
5 : en Anglais dans le texte : Hello World
6 : en Anglais dans le texte : Hello World
7 : en Anglais dans le texte : Hello World
8 : en Anglais dans le texte : Hello World
9 : en Anglais dans le texte : Hello World
10 : en Anglais dans le texte : Hello World
```

tandis qu'avec

```
# gcc -DAnglais=0 hello.c
```

[9] Personnellement, je conseille gcc -g qui permet d'utiliser le très efficace dévermineur gdb.

on obtient :

```
# a.out
0 : en Français dans le texte : Bonjour Monde
1 : en Français dans le texte : Bonjour Monde
2 : en Français dans le texte : Bonjour Monde
3 : en Français dans le texte : Bonjour Monde
4 : en Français dans le texte : Bonjour Monde
5 : en Français dans le texte : Bonjour Monde
6 : en Français dans le texte : Bonjour Monde
7 : en Français dans le texte : Bonjour Monde
8 : en Français dans le texte : Bonjour Monde
9 : en Français dans le texte : Bonjour Monde
10 : en Français dans le texte : Bonjour Monde
```

3 Où le déverminage devient nécessaire

3.1 Tableaux et chaînes de caractères

En C, un tableau à une dimension de 10 entiers sera déclaré par la syntaxe

```
int x[10];
```

Les dix cellules du tableau sont accessibles par x[0] à x[9].

Les chaînes de caractères sont ainsi représentées en C par des tableaux. Par exemple unsigned char string[1024]; définit une chaîne dont la longueur maximale est ici de 1024 caractères. En fait, par convention, toute chaîne de caractère se termine par le caractère 0 (pas le chiffre '0', mais l'entier). Seuls 1023 caractères sont donc utilisables en réalité. Il est aussi possible d'utiliser des chaînes constantes délimitées par des guillemets. "chaîne\n" est donc une chaîne comportant le mot chaîne suivi d'un retour chariot, et d'un (unsigned char) 0.

La fonction strlen(s) retourne la longueur de la chaîne s jusqu'au premier délimiteur 0. La fonction strcmp(s1,s2) permet de comparer deux chaînes. Elle retourne :

-1 si lexicographiquement s1<s2,

0 si les deux chaînes sont identiques, et

+1 si s1>s2.

La commande sprintf(string,format,variables); permet de manière similaire à la fonction printf (cf. §2.2) de formatter une chaîne de caractères. Cependant la taille de la chaîne string doit être suffisante pour stocker le résultat (y compris le dernier 0 invisible de fin de chaîne). Dans le cas contraire, on risque au mieux d'avoir des erreurs surprenantes à l'exécution, au pire de générer un débordement de buffer entraînant une vulnérabilité majeure.

3.2 Allocation dynamique de mémoire

3.2.1 Qu'est-ce qu'un pointeur ?

En C un *pointeur* est indiqué par une étoile "*". Ainsi la déclaration

```
int *x;
```

définit x comme un *pointeur* sur une valeur de type int, c'est-à-dire comme l'adresse mémoire d'un entier. L'expression *x représente donc "la valeur de type int située à l'adresse x".

Mais, comme toujours en C, **la valeur de x n'est pas initialisée**. Si on souhaite initialiser cette valeur, il faut utiliser une adresse mémoire valide, par exemple, celle d'une autre variable. Elle est obtenue par l'opérateur "&". Ainsi int y,*x; peut

permettre de définir x = &y. Cette possibilité n'est intéressante que dans le cas d'un appel de fonction. En effet, si on passe en argument d'une fonction l'adresse d'une variable, la fonction va pouvoir modifier la variable locale par l'intermédiaire de son adresse. On parle alors d'**effet de bord**.

CODE 2 : Passage par adresse

// Nous commençons par inclure le fichier standard permettant d'utiliser les entrées-sorites (en l'occurence la fonction printf

```
#include <stdio.h>
int exemple(int x, int *y) {
 x += 3;
 *y += x;
 return(x);
}
int main(void) {
 int a,b,c;
 a = 2;
 b = 1;
 c = exemple(a, &b);
 printf("a = %d b = %d c = %d\n", a, b, c);
```

// Le retour de la fonction main constitue le code retour du programme après son exécution. Ce code est accessible dans le shell en affichat la variable de shell particulière $? à l'aide de la commande **echo**.

```
 return(0);
}
```

Dans l'exemple de code 2, la sortie du programme donne "a=2 b=6 c=5". La variable b a été modifiée par l'appel de fonction : on parle alors d'*effet de bord*.

3.2.2 Allocation dynamique de mémoire

Pour pouvoir stocker en mémoire une valeur entière il faut donc explicitement demander une adresse disponible. Il existe un autre moyen d'obtenir des adresses mémoire valables, c'est le rôle de la fonction malloc(). Ainsi on pourra écrire

```
int *x;

x = (int *)malloc(sizeof(int));
```

```
*x = 2324;
```

ce qui :

1. Déclare la variable x comme un *pointeur* sur une valeur de type int.
2. Affecte à x une adresse dans la mémoire qui est réservée pour le stockage d'une donnée de taille sizeof(int), donc de type int.
3. Place la valeur 2324 à l'adresse x.

Note 7 *L'appel à* x = (int *)malloc(s) *réserve en mémoire un bloc de taille* s.

Note 8 *La syntaxe* (type)(valeur) *permet d'indiquer que* valeur *est de type* type. *Cette opération s'appelle un "*cast*" (voir note 2). Le langage* C *est en effet peu typé, c'est-à-dire qu'il est possible de considérer toute valeur comme étant de n'importe quel type. Cette possibilité est toutefois à utiliser avec prudence car elle est source d'erreur*[10] *Elle est nécessaire ici, car la fonction* malloc *retourne une adresse mémoire qui par défaut est de type* (void *).

On pourra utiliser avec profit une macro pour l'allocation dynamique de mémoire :

```
#define memalloc(ncase,tcase) ((tcase *)malloc((ncase)*sizeof(tcase)))
```

La deuxième ligne de la syntaxe précédente sera alors remplacée par la ligne plus lisible

```
x = memalloc(1,int);
```

Note 9 *Lorsqu'on n'a plus besoin de cette mémoire il convient de préciser qu'elle peut à nouveau être utilisée. On utilise pour cela* free(x). *La taille n'a pas besoin d'être précisée car elle est mémorisée lors de l'appel à* malloc(). *Dans la séquence du programme, on doit toujours rencontrer un* free() *pour tout appel réussi à* malloc(). *Ceci impose de particulièrement soigner tous les retours de fonction, y compris dans les cas d'erreur.*

Si la mémoire allouée n'est pas suffisante pour stocker l'information, on peut avoir envie d'augmenter la taille allouée. Ceci est possible par la commande realloc(oldptr, size). Cette commande retourne un pointeur sur une zone mémoire de la taille demandée. Les valeurs stockées au niveau de la zone mémoire précédemment allouée sont recopiées dans le nouveau pointeur.

De manière analogue à ci-dessus, on pourra donc créer la macro suivante :

[10] On se méfiera en particulier des conversions depuis une valeur flottante vers une valeur entière. Le résultat espéré (partie entière du nombre) n'est en effet obtenu que s'il n'y a pas de dépassement de capacité (un entier est souvent stocké sur 32 bits). Dans le cas contraire, le résultat est indéterminé.

```
#define memrealloc(old,ncase,tcase) \
  ((tcase *)realloc(old,(ncase)*sizeof(tcase)))
```

et l'utiliser pour reallouer le pointeur précédent sur une zone mémoire de trois entiers :

```
x = memrealloc(x,3,int);
```

Dans le cas où la taille d'allocation diminue seules les premières valeurs stockées restent bien entendu accessibles.

La réallocation ne nécesite pas de libérer la mémoire précedemment allouée. On peut effectuer autant d'appels à realloc que l'on souhaite avant de libérer définitiveme,nt la mémoire par free en fin d'utilisation.

Dans le cas où la taille d'allocation diminue seules les premières valeurs stockées restent bien entendu accessibles.

3.2.3 Déréférenciation

La *déréférenciation* d'un pointeur consiste à extraire la valeur sur laquelle il pointe. Comme nous l'avons dit, elle est obtenue par *x. Mais le C offre des possibilités efficaces. Il est possible d'allouer par malloc un ensemble de cellules mémoires. Ainsi on pourra écrire

```
int *x;
x = (int *)malloc(10*sizeof(int));
```

ou en utilisant notre macro ci-dessus

```
int *x;
x = memalloc(10,int);
```

pour allouer 10 entiers d'un coup. Pour accéder à chaque case, il suffit alors d'utiliser la syntaxe (*(x+i)) avec i variant de 0 à 9.

3.2.4 Équivalence tableau-pointeur

Notons tout de suite qu'il n'y a pas d'équivalence vraie entre tableau et pointeur en C. Néanmoins, un tableau est en fait un pointeur, c'est-à-dire une adresse mémoire. Plus exactement, un tableau C tel que défini section 3.1 est en fait un pointeur sur la première case du tableau. Lorsqu'on déclare int x[10]; on alloue donc 10 cellules d'entiers et la syntaxe x[i] est alors équivalente à *(x+i). x est donc une variable à part entière, qui peut être passée en paramètre à une fonction.

Il est aussi possible si on a int x[10], y[10]; d'effectuer l'affectation x = y;. Toutefois, cette affectation concerne uniquement les pointeurs et non les valeurs

stockées dans les mémoires pointées. On a donc par la suite identité physique des deux tableaux, et toute modification d'une cellule y[i] modifie identiquement x[i].

Toutefois, la distinction entre la notion de tableau et celle de pointeur apparaît sur les tableaux à double entrée. En effet considérons la déclaration int x[2][3];. Elle alloue ici 6 cellules, et on a bien x[0][0] équivalent à *(x). Mais x[i][j] est équivalent à *(x+i*3+j). Or on pourrait aussi définir un double tableau de la façon suivante

```
int *y[2],i;

for(i=0; i<2; i++)
  y[i]=memalloc(3,int);
```

Nous allouons bien ici aussi 6 cellules et nous pouvons écrire y[i][j], mais cette syntaxe est ici équivalente à *((*(y+i))+j). La variable y est cette fois-ci un double pointeur et nous allouons dans notre exemple deux cellules de plus correspondant aux deux pointeurs de chaque début de ligne du tableau y. La figure 1 montre bien cette différence d'allocation et le fait que la structure de données allouée est continue avec un tableau et discontinue avec un double pointeur.

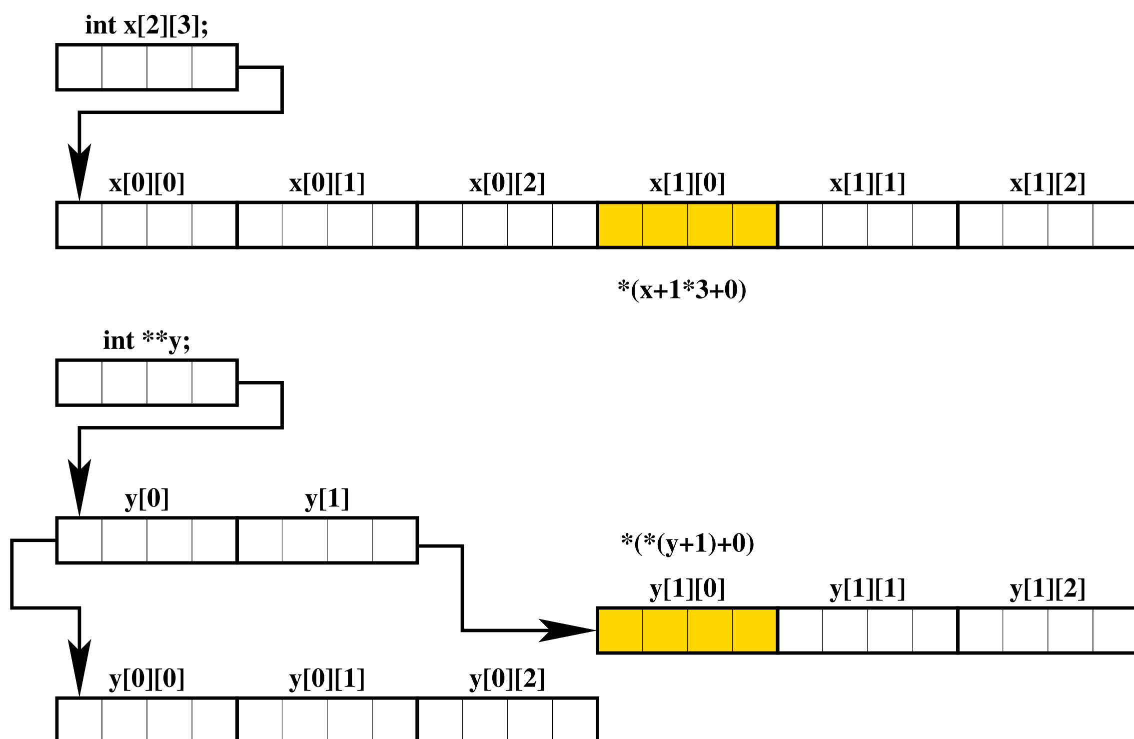


FIG. 1 – Tableau à deux entrées et double pointeur

Retenons donc qu'un tableau à une dimension est équivalent en C à un pointeur, mais que ceci est faux pour un tableau à plusieurs dimensions.

3.2.5 Pointeur de fonction

La notion de pointeur de fonction est à la base de l'implantation de beaucoup de langages objets. Elle permet de substituer dans un appel de fonctions, une fonction par une autre, du moment que les entrées-sorties de ces deux fonctions sont identiques. Le pointeur de fonction est alors l'adresse mémoire de la fonction à appeler.

Ainsi, les deux fonctions suivantes ont mêmes entrées-sortie :

```
int ajoute(short x, short y);
int retranche(short x, short y);
```

Pour déclarer une variable ou un paramètre de type pointeur sur une fonction de ce type on indiquera :

```
int (*fct)(short x, short y)
```

La variable ou le paramètre fct pourra alors être affecté à ajoute ou retranche :

```
short x,y;
int res;
int (*fct)(short x, short y);

fct=ajoute;
res=fct(x,y); /* effectue res=x+y */
fct=retranche;
res=fct(x,y); /* effectue res=x-y */
```

3.3 Gestion des options

La fonction main d'un programme peut en fait prendre des arguments main(int argc, char **argv). Ces variables correspondent au nombre d'arguments de la ligne de commande (argc) et à la liste de ces arguments sous forme de chaînes de caractères (argv[1], argv[2], ..., argv[argc-1]). La première chaîne argv[0] contient le nom d'appel du programme.

L'exemple 3 page 55 montre comment utiliser cette fonctionnalité pour passer un nom de fichier en paramètre à un programme lors de son exécution.

3.4 Gestion des fichiers

Tout ce qui suit est subordonné à l'inclusion du fichier #include <stdio.h> qui déclare le nouveau type de pointeur de fichier FILE *.

3.4.1 Ouverture de fichier

Pour ouvrir, un fichier du système **Unix**, il suffit de déclarer un pointeur de fichier FILE *pf ; et d'utiliser la fonction C

```
pf = fopen("fichier","mode");
```

Cette commande ouvre le fichier dont le nom est indiqué avec les possibilités décrites par l'un des modes suivants :

Mode	Description
r	ouvre en lecture seule.
w	ouvre en écriture seule. Si le fichier n'existait pas, il est créé, s'il existait il est écrasé.
a	comme w mais écrit à la fin du fichier s'il existait.
r+ w+ a+	comme les précédents, mais ouvre en lecture et écriture.

Trois pointeurs de fichier particuliers existent pour tout programme, correspondant aux entrées-sorties standards du programme[11] : stdin, stdout et stderr.

3.4.2 Écriture dans un fichier

Pour écrire dans un fichier, plusieurs fonctions sont disponibles. La première d'entre elle utilise le même formalisme que printf (cf. §2.2) :

```
fprintf(FILE * pf, const char * format, ...);
```

Cette commande permet de formater un affichage texte, mais il est aussi possible de placer dans un fichier des données binaires. On utilise pour cela la fonction :

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE * stream);
```

La fonction fwrite écrit un nombre nmemb de données, chacune d'elles représentés sur size octets, dans le fichier pointé par stream, après les avoir lus depuis l'emplacement pointé par ptr. La fonction fwrite renvoit le nombre d'éléments correctement écrits (et non pas le nombre d'octets). Si une erreur se produit le nombre renvoyé est plus petit que nmemb et peut même être nul.

Par exemple, pour écrire un grand nombre représenté par un tableau de 20 entiers (soit 640 bits) on pourra effectuer :

[11] En fait, cette propriété n'est assurée que pour un système d'exploitation **Unix**, même si la plupart des compilateurs C fonctionnant dans d'autres environnements tentent de simuler ce comportement

```
unsigned int GrandNombre[20];
size_t nb;
FILE *pf;

(...)

nb=fwrite(GrandNombre, sizeof(unsigned int), 20, FILE * pf);
if(nb != 20) { traitement de l'erreur }
```

3.4.3 Lecture d'un fichier

Inversement à la fonction d'écriture fwrite existe la fonction équivalente fread.

```
size_t fread (void *ptr, size_t size, size_t nmemb, FILE *stream);
```

La fonction fread lit nmemb données, chacune d'elles représentée par size octets, depuis le flux pointé par stream, et les stocke à l'emplacement pointé par ptr, qui doit bien sûr être alloué. Tout comme ci-dessu, la fonction fread renvoit le nombre d'éléments correctement lus. Si une erreur se produit le nombre renvoyé est plus petit que nmemb et peut même être nul.

Pour relire le grand nombre ci-dessus, on aurait donc à écrire de la même façon :

```
unsigned int GrandNombre[20];
size_t nb;
FILE *pf;

(...)

nb=fread(GrandNombre, sizeof(unsigned int), 20, FILE * pf);
if(nb != 20) { traitement de l'erreur }
```

L'exemple 3 page 55 utilise ces fonctions de lecture et d'écriture.

3.4.4 Autre fonctions

3.4.4.1 Accès direct au fichier

En lecture comme en écriture, les fonctions ci-dessus provoquent un avancement normalement irréversible dans le fichier. C'est-à-dire que si on fait successivement deux appels à fwrite et fprintf, les caractères générés par les deux fonctions vont apparaître successivement. De même, la lecture dans le fichier n'est pas réversible : la lecture suivante reprend à partir du premier caractère non lu. En fait, ceci n'est pas tout à fait vrai. En effet certains types de fichiers, notamment les

fichiers stockés sur disque dur, sont en accès direct. Il est donc possible de se placer à n'importe quel endroit du fichier. Les commandes principales pour gérer cet accès direct sont :

- void rewind (FILE *stream); qui replace le pointeur courant au début du fichier;
- long ftell (FILE *stream); qui indique la position courante par rapport au début du fichier;
- int fseek (FILE *stream, long offset, int whence); qui permet de replacer la position courante à la valeur positive ou négative offset calculée relativement à :
  - le début du fichier si whence vaut SEEK_SET ;
  - la position courante si whence vaut SEEK_CUR ;
  - la fin du fichier si whence vaut SEEK_END.

3.4.4.2 Lecture de fichier texte

Pour lire un fichier texte ligne par ligne, on peut aussi utiliser :

```
char * fgets (char * buffer, int sizebuf, FILE * pf);
```

Cette fonction lit un fichier texte ligne par ligne et place le résultat de cette lecture dans le tampon avec un délimiteur 0 de fin de chaîne. Si le tampon n'est pas assez grand pour stocker toute la ligne, alors seuls les sizebuf - 1 premiers caractères sont stockés. L'appel suivant à fgets stockera les caractères suivants.

L'exemple de code 7 page 69 permet de voir comment utiliser ces fonctions. On notera que l'indication de fin de fichier n'intervient qu'après qu'une lecture erronée a été tentée. C'est ce qui explique dans l'exemple de code 7 pourquoi le message d'erreur de lecture est affiché si le fichier se termine par un retour chariot. La compréhension de ce phénomène est laissée à titre d'exercice.

3.4.4.3 Fin de fichier

En cas d'erreur de lecture on peut vérifier si cette erreur correspond à la fin de fichier par la commande feof(FILE *fd);. Enfin la fonction fclose(pf); permet de fermer un fichier, en vidant auparavant son buffer.

Note 10 *Il faut savoir que le système de fichiers d'***Unix** *est bufferisé. Ainsi pour l'entrée standard, celle-ci n'est prise en compte que ligne par ligne après un retour chariot. De même pour la sortie standard. Lorsqu'on écrit dans un fichier, où que la sortie standard est redirigée, la bufferisation est encore plus importante puisque la taille du buffer peut atteindre plus de 8000 caractères. Pour forcer l'écriture, on utilise* fflush(pf);. *Cette fonction est utile, car si un programme en tâche de fond*

n'arrive pas à son terme (arrêt intempestif de la machine, ou erreur à l'exécution par exemple) les buffers sont irrémédiablement perdus.

3.5 Déverminage par gdb

3.5.1 Intérêt du déverminage

Dès que les structures de données deviennent plus complexes avec notamment des pointeurs, des erreurs de programmation sont inévitables. Ces erreurs, surtout lorsqu'elles interviennent dans la manipulation des pointeurs qui sont des adresses mémoires, peuvent conduire à des erreurs incompréhe,sibles au premier abord, dûes au fait que des modifications de la mémoire sont effectuées dans la pile du processus. Ainsi, on peut très bien observer dans le cas des débordements de tampon qu'une fonction ne revient pas à l'endroit d'où elle a été appelée. C'est même ce mécanisme qui est à l'origine de vulnérabilités par débordement de tampon ou *buffer overflow*.

Il est donc important de savoir utiliser des outils de déverminage pour trouver la cause de ces erreurs de programmation et y remédier. Dans le monde **Unix** l'outil couramment utilisé conjointement avec le compilateur **gcc** est **gdb**.

3.5.2 Compilation en mode déverminage

La première chose à faire pour utiliser cet outil de déverminage est de compiler son programme avec l'option -g. Puis on peut lancer le debugger par la commande

```
gcc -g -o [fichier exécutable] [fichiers sources]
gdb [fichier exécutable]
```

L'utilisation d'un programme de déverminage en mode texte est possible mais une interface graphique est beaucoup plus confortable. Nous utiliserons celle incluse dans **emacs** car l'ensemble de ces programmes constitue un environnement de développement suffisamment complet, mais qui reste suffisamment simple pour les programmes qui seront à développer dans le cadre de la formation du BESSSI.

3.5.3 Lancement de l'interface graphique de déverminage

Toutefois, pour pouvoir être utilisé efficacement, une interface fenêtrée est largement préférable. On prendra par exemple le puissant éditeur de fichier **emacs** et on lancera le débugger conformément à la figure 2.

On saisit ensuite dans la ligne du bas de la fenêtre, conformément à la figure 3, le nom du fichier exécutable (`a.out` puisque nous n'avons pas renommé ce fichier lors de la compilation).

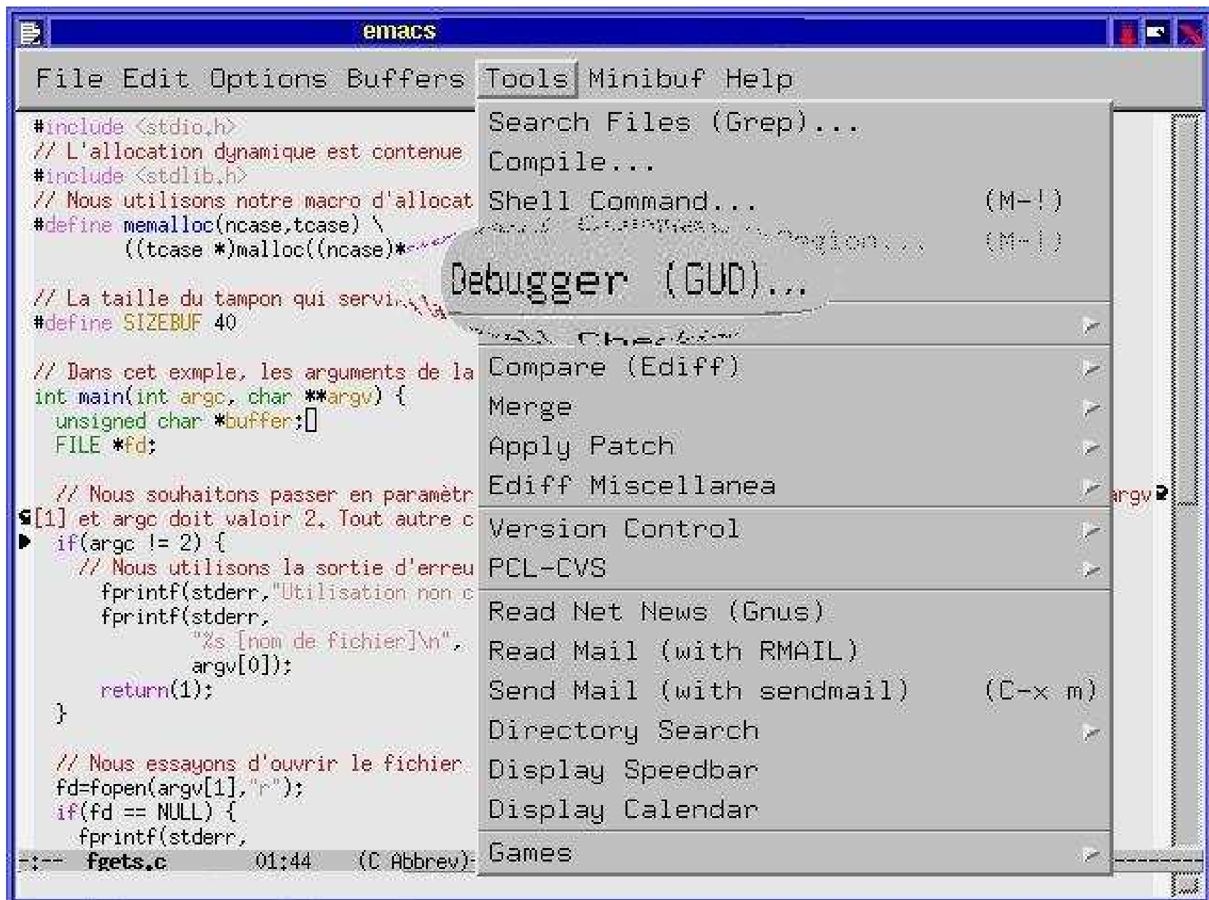


FIG. 2 – Lancement de **gdb** sous **emacs**

La fenêtre se scinde alors en deux (voir figure 4 avec une partie contenant le fichier source et l'autre contenant l'interface texte avec le programme de déverminage. Les différentes commandes de **gdb** peuvent alors être utilisées dans l'interface texte. Par exemple

```
break main
run fgets.c
```

permet de poser un point d'arrêt sur la fonction **main** et de lancer le programme avec comme argument "fgets.c" de façon similaire à ce qui serait obtenu directement par l'exécution de la commande a.out fgets.c. Le résultat de ces deux commandes est présenté figure 5. On notera que l'interface d'emacs pointe sur le fichier source l'endroit où l'exécution du programme est arrêtée. Ceci est possible grâce aux informations ajoutées à la compilation par l'option -g.

FIG. 3 – Lancement de **gdb** sous **emacs** : saisie du fichier exécutable

3.5.4 Fonctions principales de **gdb**

La première des commandes à connaître est la commande help qui permet d'entrer dans le manuel intégré de **gdb**. Le tableau ci-dessous constitue un premier guide d'utilisation avec les commandes permettant un déverminage basique d'un programme. Ces commandes sont normalement suffisantes pour déverminer un programme que l'on a soit même écrit. Les caractères soulignés constituent l'abbréviation minimale qui peut être utilisée en lieu et place de la commande.

Commande	Syntaxe	Description
break	break [nom de fonction] break [numéro de ligne]	Permet de poser un point d'arrêt soit sur une fonction soit au niveau d'une ligne du programme source. Pour poser un point d'arrêt au niveau d'une ligne de code, on peut utiliser l'interface d'emacs (raccourci clavier Ctrl-X [espace] ou menu déroulant GUD).

Commande	Syntaxe	Description
conditional	conditional [numéro de point d'arrêt] [test]	Conditionne le point d'arrêt à la satisfaction du test. La syntaxe du test est identique à celle du langage C et peut reprendre les variables du programme.
continue	continue	Continue l'exécution du programme jusqu'au prochain point d'arrêt.
delete	delete [numéro de point d'arrêt]	Efface le point d'arrêt correspondant.
display	display [variable]	Affiche à chaque étape d'exécution la valeur de la variable.
down	down	Permet de redescendre successivement les appels de fonction dans la pile d'appel (voir up).
file	file [nom d'exécutable]	Recharge la table des symboles de l'exécutable pour débugger le programme correspondant.
finish	finish	Continue l'exécution jusqu'au retour de la fonction courante.
info	info break	La commande info permet d'afficher les données internes utilisées par **gdb**. Notamment, elle permet d'afficher la liste des points d'arrêt posés.
next	next	Saute une ligne dans le programme source en poursuivant l'exécution correspondante.
print	print [variable]	Affiche la valeur de la variable. La syntaxe est identique à celle du langage C. Il est donc possible d'afficher le contenu d'un pointeur par print * ptr. Il est aussi possible d'utiliser les commandes de conversion. Par exemple pour afficher l'adresse du pointeur sous forme d'entier on pourra faire print (int) ptr.

Commande	Syntaxe	Description
run	run [arguments]	Lance le programme à déverminer avec les arguments indiqués. Ces arguments peuvent inclure des redirections vers des fichiers comme dans un shell classique. La commande run sans argument reprend les arguments de la commande précédente. Pour effacer des arguments utiliser la commande set. Pour voir les arguments mémorisés utiliser la commande show.
set	set [argument de set]	Faire help set pour voir l'ensemble des commandes utilisables. On pourra utiliser notamment set args pour effacer les arguments mémorisés dans la commande d'exécution du programme.
show	show [argument de show]	Faire help show pour voir l'ensemble des commandes utilisables. On pourra utiliser notamment show args pour voir les arguments mémorisés dans la commande d'exécution du programme.
step	step	Poursuit l'exécution du programme d'une instruction. Si celle-ci correspond à un appel de fonction, rentre dans la fonction correspondante.
undisplay	undisplay [numéro d'affichage]	Annule l'affichage correspondant.
up	up	Permet de remonter successivement les appels de fonction dans la pile d'appel.

Commande	Syntaxe	Description
where	where	Affiche la pile d'exécution du programme à l'instruction courante. En d'autres termes montre la série des appels de fonction successifs qui à conduit à l'état actuel du programme. Attention, il est courant que cette pile soit altérée par une erreur de programmation, notamment en cas de débordement de buffer.

3.5.5 Règles de programmation pour faciliter le déverminage

Un outil de déverminage fonctionnera d'autant mieux et sera d'autant plus utile que quelques règles simples seront respectées au niveau de la programmation.

1. Ne placer sur chaque ligne de programme qu'une seule instruction. En effet, dans la plupart des cas on effectuera des sauts d'instruction ligne à ligne.
2. Bien prototyper les fonctions pour que le dévermineur sache interpréter les arguments.
3. Limiter l'utilisation des macros au strict nécessaire, car comme elles sont remplacées syntaxiquement avant la compilation, le dévermineur n'a pas les moyens de remonter complètement au code source initial.

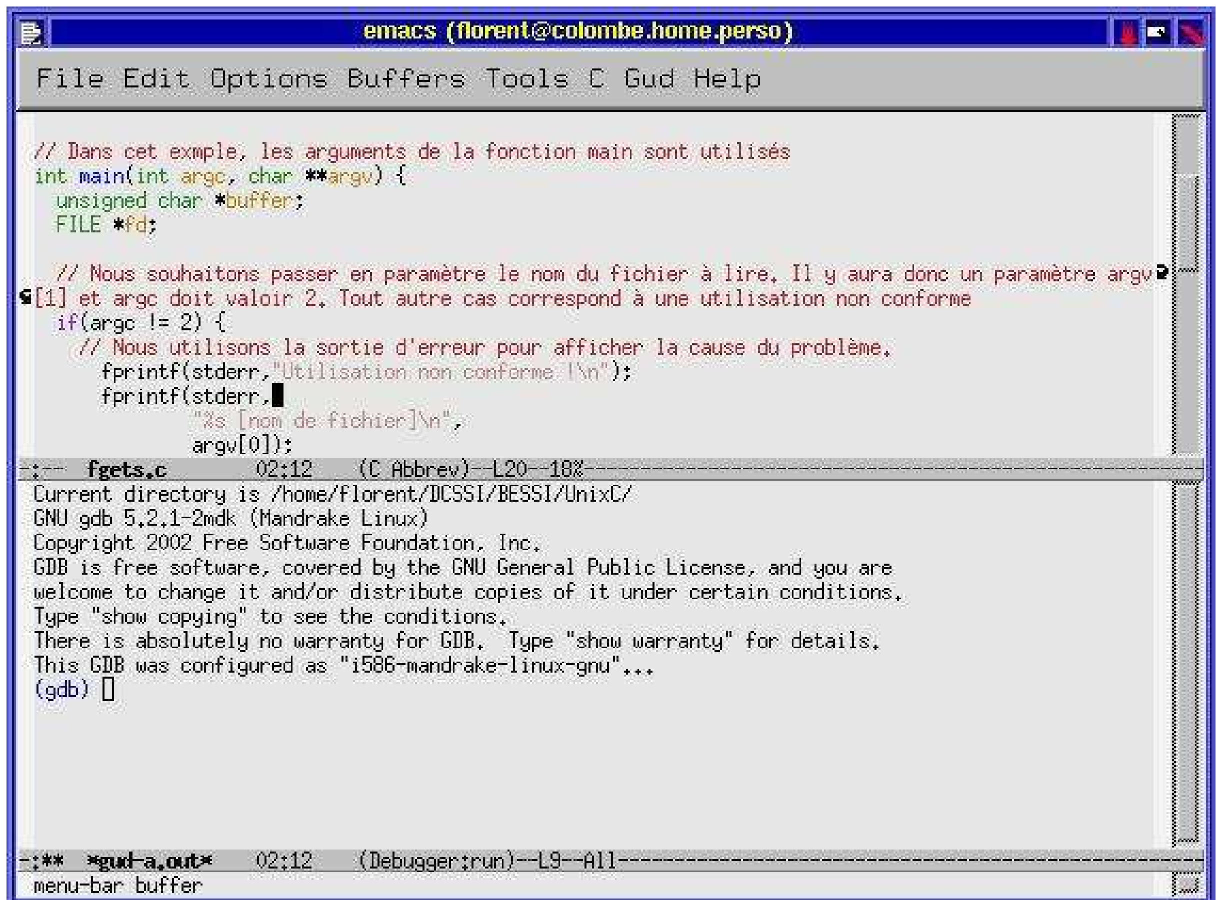


FIG. 4 – Interface de **gdb** sous **emacs**

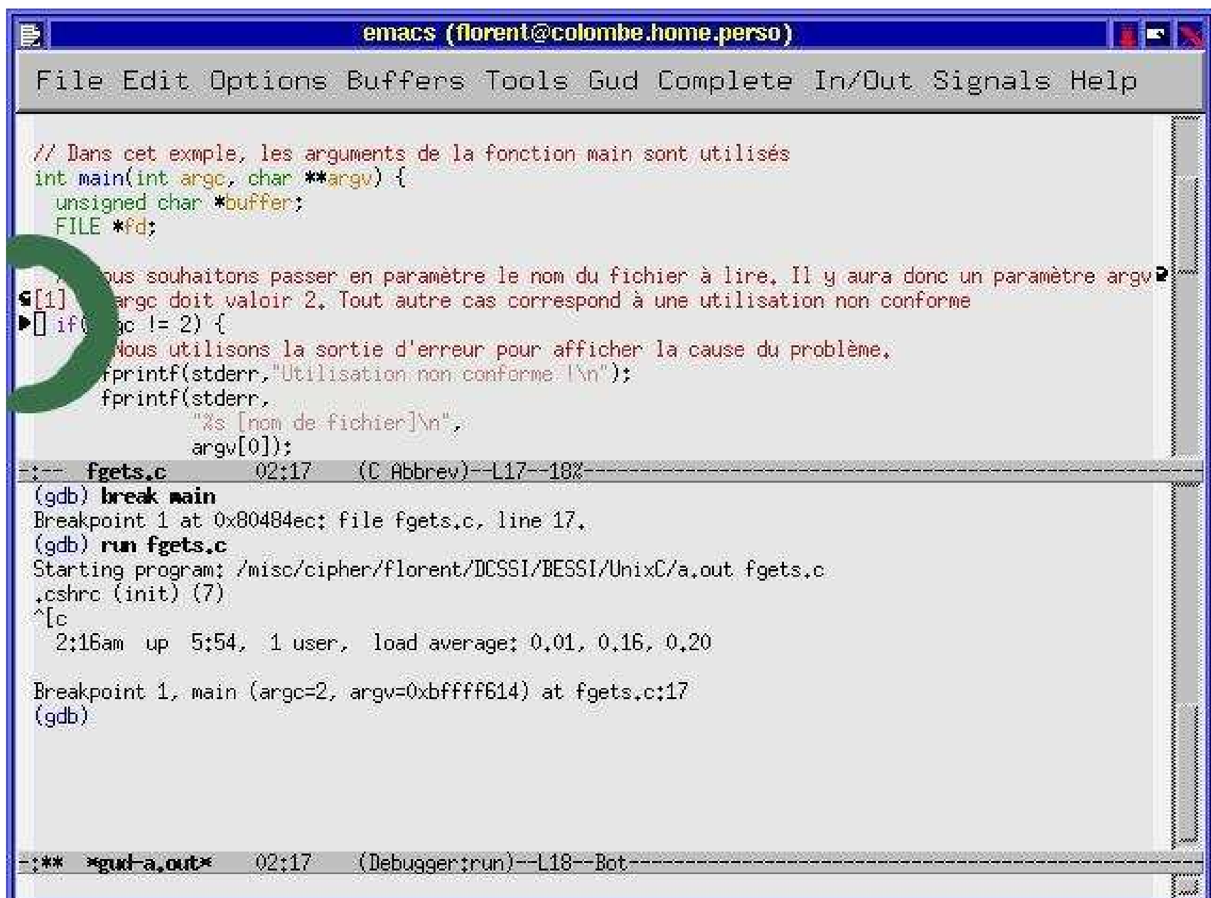


FIG. 5 – Point d'arrêt **gdb** sous **emacs**

4 Structures de données évoluées

4.1 Liste chaînées

4.1.1 Listes, piles, files, etc.

Lorsque la quantité de données à stocker ne peut être déterminée à l'avance il peut être utile de stocker celles-ci au fur et à mesure dans une structure qui s'étendra progressivement. Bien sûr une structure de type pointeur sur une zone mémoire pourrait être réallouée à chaque nouvelle entrée, mais cette opération conduirait à recopier les données déjà entrées à chaque réallocation. Il est donc préférable d'adopter une structure de données qui permette l'ajout et le retrait aisé d'une donnée sans conduire à recopier l'ensemble des données déjà entrées. Un exemple de ce type de structure de données est la liste chaînées.

Une liste chaînée d'entiers courts sera par exemple définie en C par la structure de données suivante :

```
typedef struct liste_chainee {
  short valeur;
  struct liste_chainee *p_suivant;
} * ListeChainee;
```

Cette déclaration de type permet ensuite de déclarer une variable L de type ListeChainee. Si la structure de données ainsi déclarée contient finalement trois éléments, elle peut être schématisée en mémoire conformément à la figure 6.

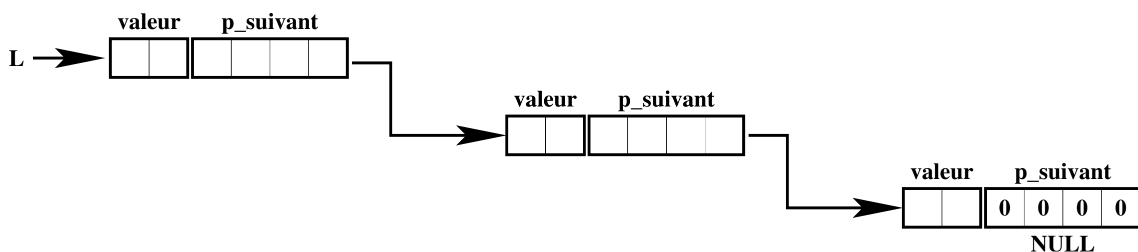
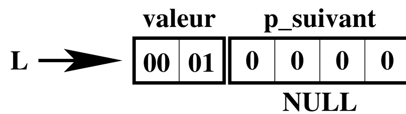


FIG. 6 – Liste chaînée de trois éléments

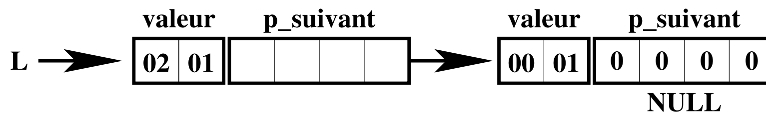
Pour atteindre cette structure, l'insertion successive des trois éléments peut se faire conformément à la figure 7. On voit sur cette figure que la fonction d'insertion utilisée réalise celle-ci en début de liste chaînée. On a ainsi créé une pile informatique. Les différents éléments ajoutés successivement s'entassent sur la pile qui pointe toujours sur le dernier élément entré. On pourrait inversement ajouter l'élément en fin de liste. On aurait alors une file. On parle aussi de liste LIFO (*last in first out* dernier entré, premier sorti) ou FIFO (*first in first out* premier entré, premier sorti).

L = NULL;

L = Insertion(0x0001, L);



L = Insertion(0x0201, L);



L = Insertion(0x1234, L);

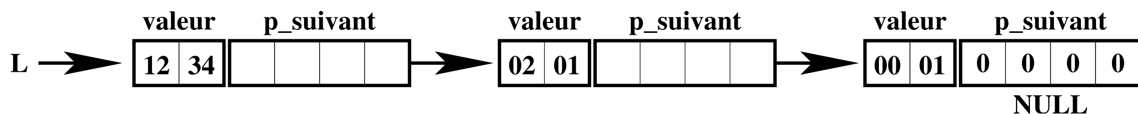


FIG. 7 – Insertion d'éléments dans une liste chaînée

4.1.2 Manipulation des structures de données

La manipulation de structures de données comme les listes chaînées n'est pas triviale. Quelques règles simples doivent être respectées.

Une structure de données un peu complexe est toujours amenée à évoluer. Il est donc impératif d'utiliser des macros ou des fonctions pour accéder au contenu de la structure de données afin de faciliter la modification éventuelle de cette structure ultérieurement. Par exemple, dans notre exemple on pourrait définir :

```
#define Valeur(L) ((L)->valeur)
#define Queue(L) ((L)->p_suivant)
```

L'avantage d'utiliser les macros est qu'elles peuvent apparaître dans la partie gauche d'une instruction d'affectation.

Les structures de données chaînées font intervenir des pointeurs. Il est beaucoup plus simple de définir des types qui soient cohérents, c'est-à-dire que `L` et `Queue(L)` sont par exemple des objets de même nature, donc de même type. Le type à définir est donc un pointeur sur une structure de données et non la structure de données elle-même. Ceci est rendu d'autant plus nécessaire pour pouvoir représenter la liste vide.

Comme toujours en `C` une variable déclarée n'est pas initialisée. On prendra donc l'habitude de toujours initialiser une structure de données de type pointeur

avec le mot clé NULL[12]. Une liste à NULL correspond à une liste vide.

4.2 Récursion et algorithmes de tri

4.2.1 Récursion

Une fonction récursive est une fonction qui directement ou non s'appelle elle-même. Cette notion est analogue à la démonstration par récurrence en mathématiques. Toute manipulation de structures de données comme les listes chaînées est une opération essentiellement récursive qui peut se résumer en deux cas :

1. Que fait la fonction lorsque la structure de données est vide ?
2. Que fait la fonction lorsque la structure de données n'est pas vide ?

Le deuxième cas correspond au cas de récurrence. Par exemple, pour l'affichage d'une liste chaînée on affichera la valeur puis la queue de liste. Ce type d'exemple est donné dans le programme 4 page 58.

La récursion n'est au départ pas intuituve. Il faut absolument la pratiquer pour s'en imprégner. C'est la raison pour laquelle tous les exercices proposés seront à réaliser à l'aide de fonctions récursives.

4.2.2 Algorithmes de tri

Un bon moyen de pratiquer la récursion est de se pencher sur le problème du tri de données. Bien que d'énoncé simple, le tri de données admet beaucoup de solutions algorithmiques différentes qui ont des performances différentes. Ceci permet d'introduire la notion de complexité d'un algorithme. Sans entrer dans le cadre formel de cette théorie, nous verrons sur ces exemples relativement simples comment une petite étude de complexité permet de prévoir l'efficacité d'un algorithme en fonction de la taille des données à traiter.

4.2.2.1 Tri par insertion

Le premier des algorithmes de tri est le tri par insertion, couramment pratiqué par tout joueur de cartes. On prend une carte puis une deuxième qu'on ordonne par rapport à la première, puis une troisième qu'on ordonne par rapport aux précédentes en l'insérant à sa place, et ainsi de suite...

En C sur un tableau de n éléments, cela donne à peu près :

```
for(i=1; i<n; i++) {
  cle = tableau[i];
  for(j = i-1; j >= 0; j--) {
```

[12] Ce mot-clé est en fait une macro qui signifie (void *)0)

```
      if(tableau[j] <= cle)
        break;
      tableau[j+1] = tableau[j];
    }
    tableau[j+1] = cle;
  }
```

Pour évaluer la complexité de cet algorithme, nous allons regarder le nombre de tests de comparaison effectué.

Note 11 *Rappelons tout d'abord la notation classique $O()$. Soient f et g deux fonctions à valeurs réelles, alors*

$$f(x) = O(g(x))$$

si et seulement si, il existe deux constantes positives N et k telles que pour tout $x > N$ on ait

$$f(x) \leq kg(x).$$

Au pire, les éléments transmis sont ordonnés de façon inverse (du plus grand au plus petit) si bien que l'insertion se fait toujours en première position (j = -1). On a donc dans ce cas à parcourir la boucle interne en entier soit un nombre total de comparaisons de :

$$\begin{aligned} \mathcal{C}_{pire}(n) &= \sum_{i=1}^{n-1} i \\ &= \frac{n(n-1)}{2} \\ &= n^2(\frac{1}{2} - \frac{1}{2n}) \end{aligned}$$

Cette dernière façon d'écrire le résultat permet de bien voir que la complexité dans le cas le pire est quadratique :

$$\mathcal{C}_{pire}(n) = O(n^2).$$

Dans le cas le meilleur ou le tableau est déjà ordonnée, il est clair que le tri par insertion ne va effectuer au contraire qu'une comparaison par boucle. On a donc :

$$\mathcal{C}_{meilleur}(n) = O(n).$$

Mais en moyenne le tri par insertion reste en $O(n^2)$ et c'est donc un algorithme peu efficace car la multiplication de la taille de l'ensemble à trier par 10 entraîne une multiplication par 100 du temps de calcul.

4.2.2.2 Tri par fusion

Le tri par fusion est une procédure récursive de tri qui tire partie du fait que le tri par insertion est bon dans le cas d'un tableau déjà trié.

Si on a deux tableaux de taille respectives $\lfloor n/2 \rfloor$ et $n - \lfloor n/2 \rfloor$ triés, alors fusionner les deux tableaux en un tableau de taille n trié peut se faire par insertion en $O(n)$.

Pour trier un tableau de taille n on va donc le scinder en 2 et procéder par dichotomie sur chaque moitié de tableau avant de fusionner les tableaux ainsi triés.

L'implantation de ce tri est l'objet de l'exercice 6.

4.3 Arbres binaires

4.3.1 Structure de données

Les arbres binaires de recherche sont un autre exemple de structures de données. Là encore, les données sont chaînées entre elles, mais elles sont aussi ordonnées par rapport à une relation d'ordre définie sur une clef.

Par exemple, on pourra définir la structure d'arbre binaire suivante :

```
typedef struct arbre_binaire {
  short clef;
  // unsigned char *chaine;
  struct arbre_binaire * p_fils_gauche;
  struct arbre_binaire * p_fils_droit;
} * ArbreBinaire;
```

Dans cet exemple, les clefs d'ordonnancement sont des entiers courts et les données stockées dans l'arbre sont des chaînes de caractères. Les données stockées sont indépendantes de l'indexation effectuées par la clef. C'est pourquoi, dans la suite, nous simplifierons la description des fonctions en omettant les données stockées.

La relation d'ordre que nous pouvons utiliser ici est la relation d'ordre naturelle sur les entiers. Le principe qu'un arbre binaire de recherche doit respecter est que par rapport à un nœud donné, tout nœud de clef supérieure à la clef du nœud doit se trouver dans son fils droit et tout nœud de clef inférieure doit se trouver dans son fils gauche. La figure 8 montre l'une des possibles constructions progressive d'un tel arbre.

4.3.2 Complexité de la recherche dans un arbre binaire ordonné

La manipulation d'arbres binaires est essentiellement réalisée à partir d'opérations récursives. Leur intérêt est de permettre un accès plus rapide aux données. En effet,

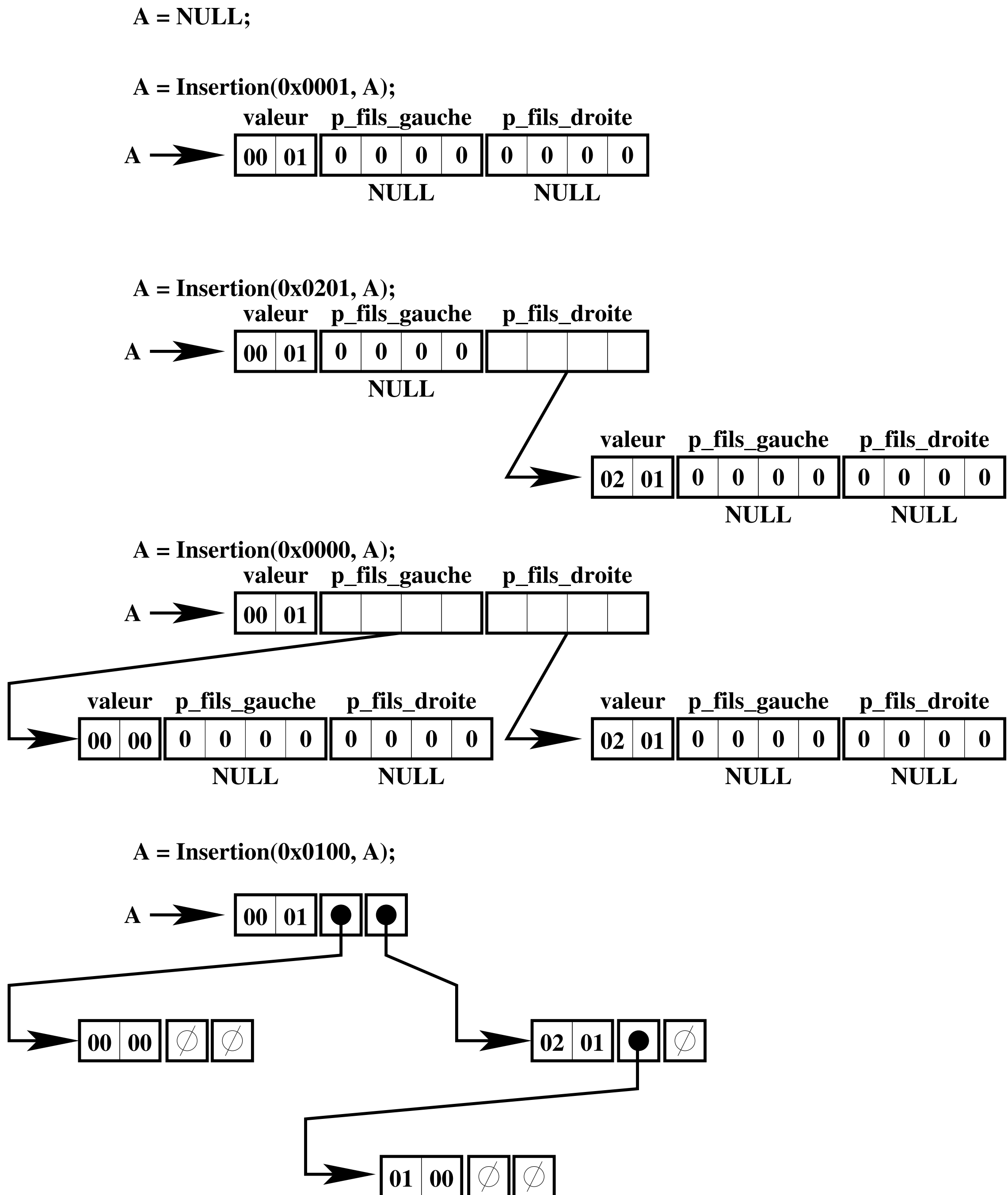


FIG. 8 – Insertion d'éléments dans un arbre binaire de recherche

si un arbre est équilibré, c'est-à-dire qu'au niveau de chaque nœud, la différence de hauteur entre les fils gauche et droit est inférieure ou égale à 1, alors un arbre de profondeur totale n peut stocker $N = 2^n$ données. Comme l'arbre est ordonné, atteindre une donnée quelconque se fait par le parcours d'un chemin de longueur

au plus $n = \log_2 N$. Le parcours d'un arbre binaire ordonné équilibré est donc au pire en $O(\log N)$. Par contre, si l'arbre n'est pas équilibré, alors le pire cas est celui d'une insertion successive de données déjà ordonnées. En effet, dans ce cas, la structure de données qui se construit est tout simplement... une liste chaînée ! Dans le cas d'un arbre binaire ordonné non équilibré, la recherche de données dans le cas le pire est donc en $O(N)$.

4.3.3 Équilibrage d'un arbre binaire

L'équilibrage d'un arbre binaire se fait de façon récursive. On commence donc par équilibrer les deux sous-arbres gauche et droit. Si après équilibrage, le nœud courant est déséquilibré, cela signifie que le nœud courant n'est pas "au milieu" des données contenues dans l'arbre. Pour rééquilibre l'arbre, l'algorithme va donc consister à déplacer la racine de l'arbre du côté où il est le plus chargé, ceci afin de répartir plus équitablement les données à droite et à gauche de la racine.

La figure 9 décrit l'algorithme sur un exemple. Si l'arbre est déséquilibré à gauche, alors on prend le fils gauche du nœud courant comme nouvelle racine. Ce faisant, le nœud courant et son fils droit doivent être raccordés à droite du plus grand des éléments situés sous la future nouvelle racine. Cette opération ne conduit pas automatiquement à un arbre équilibré du premier coup. Le rééquilibrage de la nouvelle racine est réalisé de la même façon.

Un exemple de fonction d'équilibrage d'arbre est donné en annexe dans le code 5.

4.4 Tables de hachage

La structure de données d'une table de hachage est encore un peu plus complexe. Elle emploie comme les arbres binaires une clef qui indexe les données à stocker, mais au lieu d'une fonction d'ordonnancement, on utilise une fonction de hachage $h(c)$ de cette clef qui permet pour toute valeur de la clef d'obtenir une valeur de hachage binaire de taille fixe m comprise entre 0 et $2^m - 1$.

La fonction h doit être rapide à calculer à partir de c et assurer une équirépartition des valeurs $h(c)$ dans l'espace $[, \dots, 2^m - 1]$. Comme l'espace des clés est plus grand que l'espace image de la fonction de hachage des collisions sont à prévoir. Celles-ci peuvent être gérées soit par l'emploi de listes chaînées soit par l'emploi d'arbres binaires de recherche. Dans ce dernier cas, une table de hachage est donc un tableau à 2^m entrées d'arbres binaires de recherche. Pour un espace de clefs de taille $N = 2^n$, le temps d'accès à une donnée est donc en $O(n - m)$. La table de hachage est un bon compromis temps-mémoire permettant d'accélérer notablement les recherches dans un ensemble de valeurs.

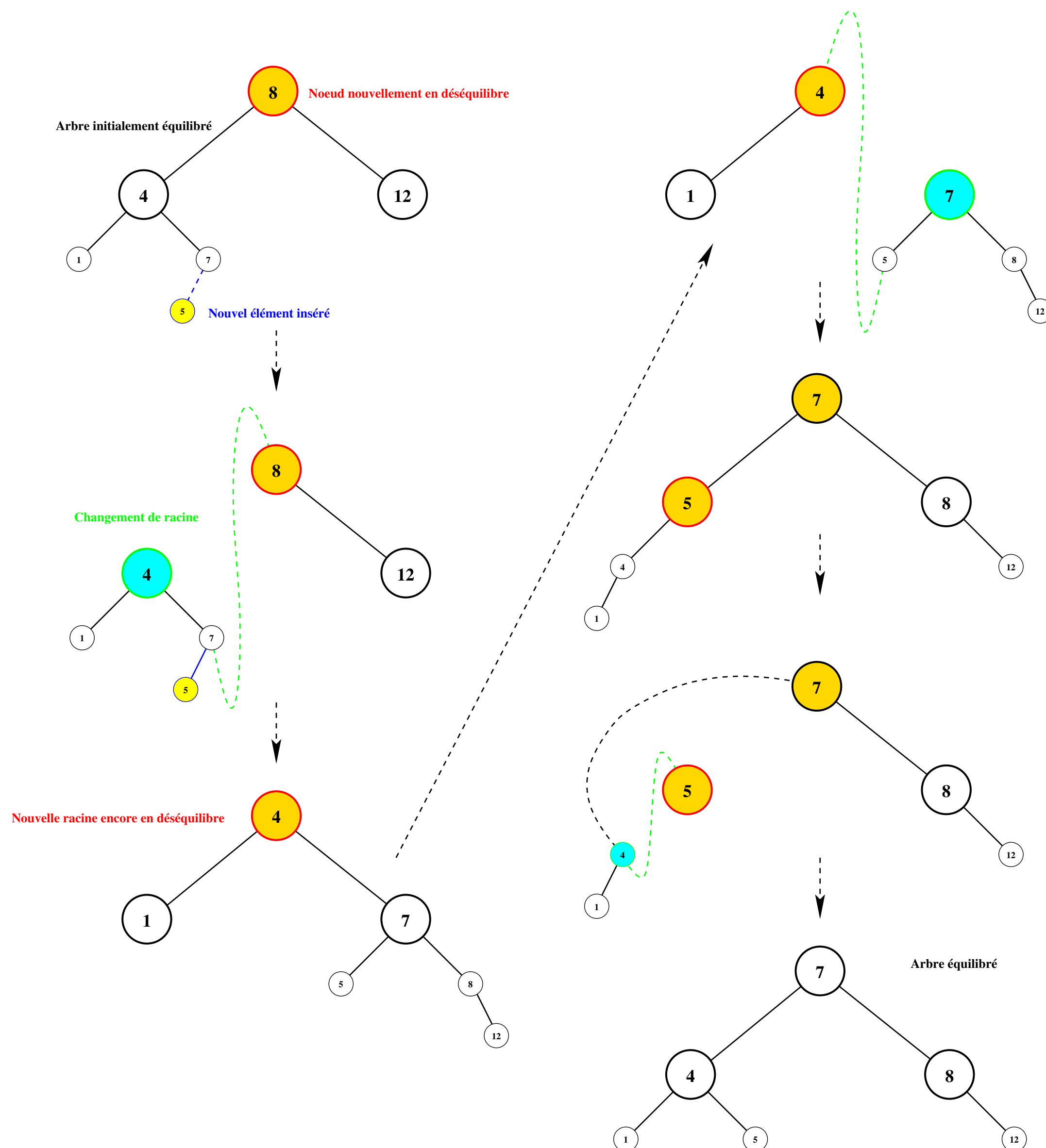


FIG. 9 – Équilibrage d'un arbre binaire

4.5 Bibliothèques et compilation partagée

4.5.1 Compilation partagée

4.5.1.1 Principe

Un des aspects de la programmation particulièrement intéressant est la programmation modulaire. Elle permet de ne pas regrouper dans un seul fichier toutes ses fonctions, mais au contraire de les répartir dans plusieurs fichiers et de les com-

piler séparément. Ceci permet, d'une part d'utiliser dans des programmes différents des opérations identiques à programmer une seule fois. D'autre part, pour les gros programmes, on pourra ainsi recompiler uniquement la partie modifiée, et non le programme tout entier. Pour cela, on s'attachera à distinguer trois types de fichiers de programmation :

1. Des fichiers contenant les fonctions communes qu'on appellera avec une extension .c, mais ne contenant que les fonctions générales et en aucun cas une fonction `main()`.
2. Des fichiers contenant les déclarations de types et de fonctions et les éventuelles macros correspondant aux fichiers précédents. On les appellera avec une extension .h, et ils seront inclus par des directives #include dans les sources des fichiers correspondants (en .c).
3. Des fichiers de programmes, avec une extension .c, incluant aussi des fichiers .h, et qui utiliseront les fonctions définies dans les fichiers .c et déclarées dans les fichiers .c.

Les variables et fonctions globales à un fichier mais locales à celui-ci seront déclarées statiques par l'emploi du mot clé static. Ce type de déclaration apparaît donc plutôt dans un fichier .c. Les variables et fonctions qui doivent pouvoir être utilisées à l'extérieur de leur fichier de définition seront déclarées externes par l'emploi du mot-clé extern. Ce type de déclaration apparaît donc plutôt dans un fichier .h, afin de pouvoir être inclus par les autres fichiers utilisateurs des fonctions et variables correspondantes.

4.5.1.2 Make

On pourra écrire avec profit un fichier `Makefile` qui permettra de compiler l'ensemble de ses fichiers de manière efficace et pratique. Par exemple, dans le fichier suivant, td.c est un fichier du type 1, td.h est un fichier de déclaration du type 2, et main.c est un fichier de programme du type 3 :

```
#-*-Makefile-*-

CC  = gcc                           # Le compilateur utilisé
OPT = -g                            # Les options à utiliser

Compil = $(CC) $(OPT)               # La commande de compilation
                                    # sera donc ici gcc -g

Test : main.o td.o                  # Test est le nom de l'exécutable
<TAB>@                              # et il a besoin des fichiers
```

```
<TAB>@                             # main.o et td.o
<TAB>$(Compil) main.o td.o -o Test # Ici la commande crééant Test.

td.o : td.c td.h                   # Les règles suivantes s'écrivent
<TAB>@                             # sur le même principe.
<TAB>$(Compil) -c td.c
main.o : main.c td.h
<TAB>$(Compil) -c main.c
```

La première ligne d'une règle indique le nom du fichier concerné suivi de " :" et de la liste des fichiers dont il dépend. Si l'un de ces fichiers est plus récent que le fichier concerné, alors les commandes de la règle situées sur les lignes suivantes sont exécutées

Note 12 *Bien noter que le premier caractère d'une ligne de commande dans un* `Makefile` *est un* `TAB`*, caractère de tabulation. Avec le fichier ci-dessus, il suffit ensuite de taper* `make Test` *(ou même* `make` *car dans ce fichier,* `Test` *est la première règle rencontrée) pour fabriquer le fichier* `Test`*.* `make` *n'exécute que les commandes de compilation nécessaires. Si seul le fichier* `main.c` *a été modifié depuis la dernière compilation, seuls* `main.o` *(puisque sa règle dépend de* `main.c`*) et* `Test` *(puisque sa règle dépend de* `main.o`*) seront recompilés. Le caractère* `@` *dans le fichier ci-dessus supprime l'affichage à l'écran de la ligne de commande. Il est nécessaire, du fait des commentaires pour faire la jonction avec la ligne de commande proprement dite car toutes les lignes de commande qui suivent une règle doivent commencer par* `TAB`*.*

4.5.1.3 Règles de programmation partagée

Pour qu'une programmation partagée puisse fonctionner correctement, notamment lors de la constitution de bibliothèques de fonctions, il est nécessaire de respecter à nouveau quelques principes simples :

1. Les fonctions exportées d'une bibliothèque doivent être correctement prototypées dans un fichier`.h`.
2. Un fichier `.h` doit obligatoirement inclure quelques lignes permettant son inclusion multiple sans erreur (cf. code 6 page 65).
3. Tous les fichiers prototypes des autres bibliothèques nécessaires à l'utilisation des fonctions de la bibliothèque doivent être inclus dans le fichier prototype.
4. La manipulation des structures de données de la bibliothèque doit pouvoir se faire soit par des macros soit par des fonctions et ne doit pas nécessiter

l'emploi direct du contenu des champs des structures de données. Ceci est particulièrement important pour permettre de faire évoluer la bibliothèque de fonction en gardant la compatibilité avec des programmes précédemment écrits.

5. En outre, il est fortement recommandé de préciser dans le fichier prototype et pour chaque fonction, variable ou macro exportée son utilité et la façon de l'employer. Pour les fonctions, on aura intérêt à adopter un formalisme précisant :
   - les entrées de la fonction, avec la description précise des paramètres attendus ;
   - les hypothèses faites sur les paramètres (par exemple, pour un pointeur s'il doit être alloué ou non) ;
   - la sortie de la fonction ;
   - les effets de bords de la fonction, c'est-à-dire toutes les modifications effectuées par la fonction sur les variables globales ou les éléments qui ont été passés par adresse (notamment les modifications à l'intérieur des structures de données).
6. *Last but not least*, un programme de test des fonctions de la bibliothèque sera très utile pour permettre son évolution tout en vérifiant que les modifications apportées n'entraînent pas de régression.

Le code 6 page 65 présente un exemple de bonnes pratiques pour un fichier prototype d'une bibliothèque de fonctions de manipulation d'arbres binaires.

4.5.2 Fabrication d'une bibliothèque

Dans le processus de compilation, une bibliothèque de fonctions peut être associée à l'exécutable de façon statique ou dynamique. Les bibliothèques dynamiques[13] sont chargées au moment de l'exécution du programme et même au moment de la première utilisation de l'une de leurs fonctions. Elles permettent une grande souplesse de mise à jour, à condition que les règles ci-dessus soient respectées et que les interfaces de la bibliothèque ne soient pas modifiées[14].

Toutefois, les bibliothèques dynamiques étant plus délicates à mettre en œuvre, nous nous contenterons dans cette introduction au C de réaliser une biblithèque statique. Sous **Unix** un fichier de bibliothèque est une archive en .a contenant les fichiers .o issus de la compilation partagée des fichiers .c. Pour constituer la bibliothèque proprement dit, on utilise la commande ar :

[13] Le terme de librairie dynamique est une mauvaise traduction de l'anglais *dynamically loaded library (DLL)*.

[14] On définit souvent les API ou *Application Programming Interface* d'une bibliothèque. Ceci correspond peu ou prou à un fichier prototype correctement documenté.

```
ar -crs [mon fichier .a] [les fichiers .o]
```

Le fichier ainsi obtenu est utilisable dans une commande de compilation à l'egal d'un fichier `.o`. Il contient l'ensemble des fonctions compilées de façon relogeable, c'est-à-dire que toutes les adresses utilisées dans ces fichiers sont des adresses relatives. L'exécutable qui résultera de l'édition de liens finale (qui consiste à associer les sauts de fonctions vers des adresses réelles dans l'exécutable) ne dépendra plus du fichier `.a` et pourra être exécuté en l'absence de celui-ci.

Annexes

A Exemples de programmes

CODE 3 : Lecture d'un fichier par fread

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

// L'allocation dynamique est contenue dans la librairie standard suivante

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

// Nous utilisons notre macro d'allocation pour plus de lisibilité

```
#define memalloc(ncase,tcase) \
        ((tcase *)malloc((ncase)*sizeof(tcase)))
```

// La taille du tampon qui servira à la lecture

```
#define SIZEBUF 40
```

// Dans cet exemple, les arguments de la fonction main sont utilisés

```
int main(int argc, char **argv) {
  unsigned char *buffer;
  FILE *fd;
```

// Nous souhaitons passer en paramètre le nom du fichier à lire. Il y aura donc un paramètre argv[1] et argc doit valoir 2. Tout autre cas correspond à une utilisation non conforme

```
  if(argc != 2) {
```

// Nous utilisons la sortie d'erreur pour afficher la cause du problème.

```
      fprintf(stderr,"Utilisation non conforme !\n");
      fprintf(stderr,
              "%s [nom de fichier]\n",
              argv[0]);
      return(1);
  }
```

// Nous essayons d'ouvrir le fichier passé en paramètre

```
  fd=fopen(argv[1],"r");
  if(fd == NULL) {
    fprintf(stderr,
            "Impossible d'ouvrir le fichier %s\n",
            argv[1]);
    return(2);
  }
```

// Nous essayons ensuite d'allouer notre tampon

```
  buffer=memalloc(SIZEBUF,unsigned char);
  if(buffer == NULL) {
```

// Dans ce cas d'erreur, il faut penser à refermer le fichier ouvert

```
    fclose(fd);
    fprintf(stderr,
            "Impossible d'allouer %d caractères\n",
            SIZEBUF);
    return(3);
  }
```

// Tant que la fin de fichier n'est pas atteinte, nous recopions sur la sortie standard les buffers lus

```
  while(!(feof(fd))) {
```

// Nous pouvons déclarer dans un bloc une variable locale à celui-ci

```
    int lus;
```

// fread retourne le nombre de carcatères lus

```
    lus=fread(buffer, sizeof(char), SIZEBUF, fd);
    fwrite(buffer, sizeof(char), lus, stdout);
  }
```

```
// Il ne faut pas oublier la libération des structures allouées

  fclose(fd);
  free(buffer);

// Le retour d'un programme sans erreur est normalement 0

  return(0);
}
```

CODE 4 : Création et affichage d'une file

```
#include <stdio.h>
#include <stdlib.h>
#define memalloc(ncase,tcase) \
        ((tcase *)malloc((ncase)*sizeof(tcase)))

// Nous définissons notre structure de liste chaînée

typedef struct liste_chainee {
  short valeur;
  struct liste_chainee *p_suivant;
} * ListeChainee;

// Ainsi que les macros d'accès à la structure de données

#define Valeur(L) ((L)->valeur)
#define Queue(L) ((L)->p_suivant)

// Notre première fonction fabrique une cellule de la liste et renseigne les
champs de la structure

ListeChainee CreeCellule(short valeur, ListeChainee L) {

// Nous allons allouer la cellule dans la variable ret

  ListeChainee ret;

// Attention nous allouons bien une taille mémoire correspondant à la cellule
et non une liste chaînée qui est un pointeur

  ret=memalloc(1,struct liste_chainee);

// Il ne faut pas oublier le cas où l'allocation ne fonctionne pas

  if(ret == NULL) {
    fprintf(stderr,"Erreur d'allocation\n");
    exit(1);
  }
```

// Reste à renseigner les champs de la structure de données ainsi allouée

```
  Valeur(ret)=valeur;
  Queue(ret)=L;
  return(ret);
}
```

// Notre deuxième fonction insère de façon récursive une valeur en fin de liste

```
ListeChainee InsereFinListe(short valeur, ListeChainee L) {
  ListeChainee ret;
```

// Comme toujours dans une fonction récursive, la première chose à faire est le test d'arrêt c'est-à-dire le cas initial lorsque la fonction est appelée avec une structure de données vide.

```
  if(L == NULL) {
```

// Si la liste est vide, insérer une valeur revient simplement à créer la première cellule et à la renseigner. C'est justement l'objet de la fonction précédente.

```
    ret=CreeCellule(valeur,NULL);
```

// Le cas d'erreur a déjà été traité donc il n'y a rien d'autre à effectuer.

```
  } else {
```

// Nous voici désormais dans le cas de récurrence. Si la liste n'est pas vide, alors comme nous insérons des données en fin de liste, nous savons que c'est la valeur non vide `L` *qui restera à retourner.*

```
    ret=L;
```

// D'autre part, il nous faut insérer la valeur en fin de liste, c'est-à-dire à la fin de la queue de la liste courante.

```
    Queue(L)=InsereFinListe(valeur,Queue(L));
  }
  return(ret);
}
```

// Notre troisième fonction affiche récursivement les éléments de la liste

```
void AfficheListe(ListeChainee L) {
```

// Là encore, la première chose à faire est le test d'arrêt c'est-à-dire le cas initial lorsque la fonction est appelée avec une structure de données vide. En l'occurence il n'y a rien à faire.

```
  if(L == NULL) {
```

// En fait, pour que l'affichage soit propre on va quand même insérer un retour chariot.

```
    printf("\n");
    return;
  }
```

// Sinon, on va afficher la valeur courante suivie de la queue de liste.

```
  printf(" %d .",Valeur(L));
  AfficheListe(Queue(L));
}
int main(int argc, char **argv) {
  int i;
  short val;
  ListeChainee L=NULL;
```

// On va insérer 10 valeurs aléatoires

```
  for(i=0; i<10; i++) {
```

// lrand48() retourne un entier sur 32 bits, alors on le réduit modulo 2^{16}.

```
    val=(short)(lrand48()%(1<<16));
```

// On insère la valeur

```
    L=InsereFinListe(val, L);
```

// Puis on affiche le résultat

```
    printf("%d : ",i);
    AfficheListe(L);
  }
  return(0);
}
```

CODE 5 : Équilibrage d'un arbre binaire de recherche

// Pour implanter un équilibrage d'arbre, il est nécessaire de modifier la structure de données pour y incorporer une information sur la hauteur de chaque nœud.

```
typedef struct arbre_binaire {
  short clef;
  int hauteur;
```

*// unsigned char *chaine ;*

```
  struct arbre_binaire * p_fils_gauche;
  struct arbre_binaire * p_fils_droit;
} * ArbreBinaire;
```

// Nous introduisons aussi des macros d'accès désormais habituelles.

```
#define ArbreClef(pA) ((pA)->clef)
#define ArbreHauteur(pA) ((pA)->hauteur)
#define ArbreFilsG(pA) ((pA)->p_fils_gauche)
#define ArbreFilsD(pA) ((pA)->p_fils_droit)
```

// La fonction d'équilibrage de l'arbre aura pour effet de bord de calculer les informations de hauteur d'arbre. Il est aussi utile d'adopter la convention que si la hauteur enregistrée dans l'arbre est nulle, alors c'est que le nœud n'a pas été équilibré. Ceci permet en effet de ne pas rééquilibrer inutilement un arbre déjà équilibré. De façon récursive on pourra alors écrire la fonction ainsi :

```
ArbreBinaire Equilibrer(ArbreBinaire pA) {
  int hg,hd;
```

// Si l'arbre est vide, il n'y a rien à effectuer.

```
  if(pA == NULL)
    return(NULL);
```

// Si la hauteur est renseignée, alors l'arbre est équilibré.

```
  if(ArbreHauteur(pA))
    return(pA);
```

// On commence par équilibrer les deux fils du nœud courant

```
  ArbreFilsG(pA)=Equilibrer(ArbreFilsG(pA));
  ArbreFilsD(pA)=Equilibrer(ArbreFilsD(pA));
```

// On calcule ensuite les hauteurs gauche et droite.

```
  hg=hd=0;
  if(ArbreFilsG(pA)) hg=ArbreHauteur(ArbreFilsG(pA));
  if(ArbreFilsD(pA)) hd=ArbreHauteur(ArbreFilsD(pA));
```

// Si l'arbre est déséquilibré...

```
  if((hd-hg < -1)||(hd-hg > 1)) {
    ArbreBinaire pnew,plink;
```

// ... nous changeons de racine en prenant celle du fils le plus long. Par voie de conséquence, le pointeur correspondant est forcément non vide.

```
    if(hg > hd) {
      pnew=ArbreFilsG(pA);
```

// On rompt le lien correspondant.

```
      ArbreFilsG(pA)=NULL;
```

// Ce faisant, le nœud n'est plus équilibré.

```
      ArbreHauteur(pA)=0;
```

// On recherche le plus grand des éléments situés sous la nouvelle racine. **Attention**, *cette fonction doit avoir pour effet de bord de remettre à zéro toutes les hauteurs des nœuds qu'elle parcourt. En effet, nous allons insérer l'ancienne racine au bout de ce parcours et l'équilibrage n'est pas conservé.*

```
      plink=ArbreMax(pnew);
```

// On relie l'ancienne racine à cet élément. Comme c'est le plus grand des éléments, le fils droit est en effet obligatoirement vide.

```
      ArbreFilsD(plink)=pA;
    } else {
```

// le cas symétrique est laissé au lecteur...

```
    }
```

// Le nouvel arbre n'est pas forcément équilibré, il convient donc de rappeler récursivement la fonction.

```
    pA=Equilibrer(pnew);
    return(pA);
  }
```

// Si l'arbre est équilibré alors il ne reste qu'à calculer la nouvelle hauteur et à la sauvegarder.

```
  if(hd>hg)
    ArbreHauteur(pA)=hd+1;
  else
    ArbreHauteur(pA)=hg+1;
  return(pA);
}
```

CODE 6 : Exemple de fichier prototype

```
// Protection contre les inclusions multiples du fichier
#ifndef __MA_VARIABLE_PERSO__
#define __MA_VARIABLE_PERSO__ 1

// Inclusion des fichiers déclaratifs des bibliothèques utilisées

#include <stdlib.h>

// Définition des types utilisés par la bibliothèque

typedef enum { OK, ERREUR } err;
typedef int clef;
typedef unsigned char * donnee;
typedef struct arbre_binaire {
  clef c;
  donnee d;
  struct arbre_binaire * p_fils_gauche;
  struct arbre_binaire * p_fils_droit;
} * ArbreBinaire;

// Définition des macros de manipulation des champs de structure

#define Clef(A) ((A)->c)
#define Valeur(A) ((A)->d)
#define FilsG(A) ((A)->p_fils_gauche)
#define FilsD(A) ((A)->p_fils_droit)

// Déclaration des variables globales exportées

extern err ArbreBinaireError;
```

```
// Prototypage et documentation des fonctions

/* Insertion d'un élément dans un arbre.
   Entrée :     une clef, une donnée et un arbre binaire A.
   Hypothèses : la donnée est une chaîne de
                caractères allouée.
   Sortie :     l'arbre binaire résultant de l'insertion
                dans A de la donnée d référencée par la
                clef c.
   Effets de bord : Si la clef c est déjà présente, alors
                l'arbre retourné est inchangé et la donnée
                correspondante n'est pas insérée. Sinon,
                une allocation de mémoire est réalisée pour
                créer le noeud correspondant. Si cette
                allocation échoue, la variable globale
                ArbreBinaireError est mise à ERREUR.
*/
extern ArbreBinaire InsertionArbre(clef c,
                                   donnee d,
                                   ArbreBinaire A)

// etc...

#endif

// Fin du fichier
```

B Exercices de programmation en C

B.1 Syntaxe élémentaire

Exercice 1 *Écrire, compiler et exécuter un programme affichant les cent premiers nombres entiers à l'exception des nombres divisibles par 13.*

B.2 Allocation dynamique de mémoire

Exercice 2 *En utilisant fgets, écrire, compiler et exécuter un programme capable de lire un fichier passé en paramètre du programme ligne par ligne, c'est-à-dire qu'un buffer contenant la ligne courante sera affiché par la commande* printf *avec le numéro de ligne le précédant.*

Exercice 3 *Lancer le programme de déverminage* **gdb** *dans l'interface* **emacs** *sur le programme de l'exercice 3 et utiliser toutes les commandes listées au paragraphe 3.5.*

Exercice 4 *Écrire un programme permettant d'insérer dans un tableau en position aléatoire une valeur entière elle même aléatoire. On utilisera la fonction de génération d'aléa* lrand48*. On effectuera pour chaque insertion une réallocation du tableau.*

B.3 Structures de données

Exercice 5 *Écrire un programme permettant d'insérer dans une liste chaînée en position aléatoire une valeur entière elle même aléatoire. Comparer le temps d'exécution de ce programme avec celui de l'exercice 4.*

Exercice 6 *Programmer récursivement le tri par insertion et le tri par fusion sur les listes chaînées de l'exercice précédent et comparer les temps d'exécution. Quelle est la complexité de l'algorithme de tri par fusion dans le cas le pire ?*

Exercice 7 *Programmer une bibliothèque de fonctions de manipulation de structures de données génériques. On pourra implanter successivement :*

- *les listes chaînées,*
- *les arbres binaires ordonnés,*
- *les tables de hachage.*

On utilisera des pointeurs de fonction pour permettre la manipulation de tout type de données.

C Corrections commentées

Quelques uns des exercices précédents sont corrigés ci-dessous. Rappelons toutefois que seule la pratique de la programmation est susceptible de permettre la compréhension de notions comme la récursion ou les pointeurs. Il est donc important de s'astreindre à faire ces exercices avant de consulter le corrigé.

CODE 7 : Lecture d'un fichier ligne à ligne par fgets

```
// Cette librairie contient les commandes malloc
#include <stdlib.h>

// Celle-ci les commandes printf

#include <stdio.h>

// Celle-ci les gestions de chaîne

#include <string.h>

// Nous définisssons une structure de données associant la chaîne de caractères tampon et sa taille allouée

typedef struct tampon {
  char * p_buf;
  int taille;
} buffer;

// Nous définissons aussi les macros d'accès associées

#define Ptr(buf) ((buf).p_buf)
#define Taille(buf) ((buf).taille)

// Nous définissons aussi nos macros d'allocation et de réallocation

#define memalloc(n,type) \
  ((type *)malloc((n)*sizeof(type)))
#define memrealloc(oldptr,n,type) \
  ((type *)realloc(oldptr,(n)*sizeof(type)))
```

// Nous définissons un type particulier pour gérer les erreurs

```
typedef enum {OK, KO, NOEOLN, LINELONG} erreur;
```

// Les prototypes des fonctions sont les suivants

```
void lecture_fichier(FILE *pf);
erreur lecture_ligne(FILE *pf, buffer *pbuf);
erreur reallocate(buffer * p_buf);
```

// La fonction de réallocation double la taille du pointeur alloué précédemment.

```
erreur reallocate(buffer * p_buf) {
  char *old;
```

// Le pointeur old *sert à stocker la valeur initiale pour permettre la libération de la mémoire en cas de problème de réallocation.*

```
  old = Ptr(*p_buf);
```

// La réallocation est effectuée ici.

```
  Ptr(*p_buf) = memrealloc(old,2*Taille(*p_buf),char);
```

// Si elle se passe mal, on libère toute la mémoire et on retourne un code d'erreur KO.

```
  if(Ptr(*p_buf) == NULL) {
    free(old);
    return KO;
  }
```

// Sinon, on mémorise la nouvelle taille allouée

```
  Taille(*p_buf) *= 2;
```

// Et on retourne le code OK comme quoi tout s'est bien passé.

```
  return OK;
}
```

// La fonction de lecture prend en paramètre le fichier à lire et la structure de tampon à utiliser pour la lecture. Cette dernière est passée par adresse pour pouvoir être réallouée. La fonction lecture_ligne *retourne :*

OK si la lecture d'une ligne a bien été effectuée et que le tampon contient cette ligne complète ;

KO si aucune lecture n'a pu être effectuée (fin de fichier vraisemblable) ;

LINELONG si une erreur de réallocation s'est produite (ligne trop longue) ;

NOEOLN si une lecture s'est produite mais qu'une erreur est intervenue avant d'avoir pu compléter la ligne (il est vraisemblable que la dernière ligne soit sans caractère de fin de ligne à la fin du fichier).

```
erreur lecture_ligne(FILE *pf, buffer *p_buf)
{
  char *retour;
  int longueur;
```

// La variable locale retour *permet de savoir si la lecture s'est bien passée. La lecture se fait dans le tampon pour au maximum la taille allouée.*

```
  retour=fgets(Ptr(*p_buf),Taille(*p_buf),pf);
  if (retour==NULL)
    return KO;
```

// La longueur de la chaîne lue va nous permettre de savoir si nous avons atteint les limites du tampon ou non.

```
  longueur=strlen(Ptr(*p_buf));
```

// En effet si le dernier caractère de la chaîne est un retour chariot, alors nous avons lu la ligne. Sinon, il faut réallouer notre tampon et continuer la lecture de la ligne.

```
  while (Ptr(*p_buf)[longueur-1] != '\n') {
```

// Si le retour de reallocate n'est pas nul (c'est-à-dire est différent de OK) alors c'est qu'une réallocation s'avère impossible.

```
    if(reallocate(p_buf))
      return LINELONG;
```

// Attention toutefois à ne pas écraser les données déjà lues et stockées dans le tampon. Nous avons donc à décaler le pointeur de la quantité de données déjà lues pour stocker les nouvelles données à la suite des précédentes. La taille de lecture dans le buffer s'en trouve réduite d'autant.

```
    retour=fgets(Ptr(*p_buf)+longueur,
 Taille(*p_buf)-longueur,
 pf);
```

// En cas d'erreur, on est à la fin du fichier.

```
    if (retour==NULL)
      return NOEOLN;
```

// Sinon on recalcule la longueur de chaîne pour savoir à nouveau si on est en fin de ligne.

```
    longueur=strlen(Ptr(*p_buf));
  }
  return OK;
}
void lecture_fichier(FILE *pf)
{
  buffer buf;
  int fin_de_fichier=1;
```

// Pour vérifier l'adéquation de la taille du tampon à la taille des lignes lues, nous allons mémoriser la longueur maximale d'une ligne

```
  unsigned int len,maxlen=0;
```

// Nous initialisons le tampon avec une taille volontairement ridicule de 2 caractères. Malgré cela, le mécanisme de réallocation va nous donner une taille de tampon correcte. La taille de 1 caractère ne fonctionne pas car la fonction fgets *a besoin d'au moins un caractère pour stocker le délimiteur nul de fin de chaîne.*

```
  Taille(buf)=2;
  Ptr(buf)=(char *)malloc(Taille(buf)*sizeof(char));
```

// Bien que l'allocation de deux caractères soit probable, nous prenons quand même la précaution de tester le cas d'erreur.

```
  if(Ptr(buf)==NULL)
    {
      fprintf(stderr,"Déjà plus de mémoire !\n");
      return;
    }
```

// Reste ensuite à lire le fichier ligne à ligne.

```
  do {
    fin_de_fichier=lecture_ligne(pf, &buf);
```

// ...et à afficher la chaîne mémorisée dans le tampon en fonction du code d'erreur retourné.

```
    switch(fin_de_fichier) {
    case NOEOLN:
      printf("%s\n",Ptr(buf));
      fprintf(stderr,"Fichier sans fin de ligne.\n");
      break;
    case LINELONG:
      fprintf(stderr,"Impossible d'allouer un tampon"
      " de taille double de %d.\n",Taille(buf));
      break;
    case KO:
```

// Pas de lecture effectuée, donc pas d'affichage

```
      break;
    case OK:
```

// Dans le cas général on n'affiche pas de caractère de fin de ligne car il est inclus dans le tampon.

```
      printf("%s",Ptr(buf));
      break;
    }
```

// S'il n'y a pas eu problème d'allocation (qui libère le pointeur et le rend inutilisable) on vérifie que la longueur de ligne n'est pas un nouveau maximum

```
    if((fin_de_fichier == OK)
      ||(fin_de_fichier == NOEOLN)) {
      len=strlen(Ptr(buf));
      if(len>maxlen)
maxlen=len;
    }
```

// Cette boucle se poursuit tant que la fin de fichier n'est pas atteinte.

```
  } while (fin_de_fichier == OK);
```

// Pour info, nous indiquons la taille allouée du tampon et la taille de la plus longue ligne du fichier.

```
  fprintf(stderr,"Info : taille de tampon %d "
  "(max. %d).\n",Taille(buf),maxlen);
```

// Et nous n'oublions pas de libérer le pointeur interne au tampon.

```
  free(Ptr(buf));
}
```

// La fonction principale traite des arguments du programme.

```
int main(int argc, char **argv)
{
  FILE *pf;
```

// On commence par tester le nombre de paramètres passés sur la ligne de commande. Il doit n'y en avoir qu'un le nom de fichier.

```
  if(argc != 2){
```

// Si ce n'est pas le cas nous rappelons l'utilisation du programme

```
    fprintf(stderr, "%s [nom de fichier]\n",argv[0]);
    return KO;
  }
```

// Nous ouvrons alors le fichier indiqué en paramètre sur la ligne de commande en lecture

```
  pf=fopen(argv[1],"r");
```

// Nous traitons le cas d'erreur

```
  if(pf==NULL){
    fprintf(stderr, "Erreur d'ouverture du fichier"
    " %s\n", argv[1]);
    return KO;
  }
```

// Nous appelons la fonction de lecture

```
  lecture_fichier(pf);
```

// Nous refermons la structure de lecture du fichier.

```
  fclose(pf);
  return OK;
}
```

CODE 8 : Tri par insertion et par fusion

```
#include <stdlib.h>
#include <stdio.h>
#define memalloc(n,type) \
        ((type *)(malloc((n)*sizeof(type))))
```

// Nous définissons un type particulier Donnee *pour les données*

```
typedef int Donnee;
```

// Le maillon d'une liste chaînée est constitué d'une donnée et d'un pointeur sur l'élément suivant de la liste chaînée.

```
struct maillon {
  Donnee data;
```

// Le pointeur pSuivant *est de type* pListe *selon la définition ci-dessous.*

```
  struct maillon * pSuivant;
};
```

// Le type pListe *que nous définissons ci-dessous est un pointeur pour qu'une liste et sa queue aient même type* pListe*.*

```
typedef struct maillon *pListe;
```

// Les macros d'accès aux champs de la liste sont écrites pour le type pListe

```
#define Valeur(pL) ((pL)->data)
#define Queue(pL) ((pL)->pSuivant)
```

// Nous déclarons ensuite les prototypes des fonctions implantées.
// La fonction de construction alloue une structure de données de type struct maillon*, la renseigne avec les deux éléments* d *et* pL *et retourne l'adresse de ce maillon.*

```
pListe Constructeur(Donnee d, pListe pL);
```

// Les deux fonctions suivantes réalisent respectivement l'ajout en première position et en dernière position d'une donnée `pL`

```
pListe AjoutPile(Donnee d, pListe pL);
pListe AjoutFile(Donnee d, pListe pL);
void Imprime(pListe pL);
void ImprimeReverse(pListe pL);
```

// Fonction de construction

```
pListe Constructeur(Donnee d, pListe pL) {
  pListe pmailloninit;
```

// allocation d'un maillon

```
  pmailloninit=memalloc(1,struct maillon);
```

// test du cas d'erreur

```
  if(pmailloninit == NULL)
    exit(1);
```

// affectation des paramètres de la fonction...

```
  Valeur(pmailloninit)=d;
```

// ...notamment le chaînage avec la liste pL.

```
  Queue(pmailloninit)=pL;
  return pmailloninit;
}
```

// L'ajout dans une pile ne fait rien de plus que l'opération précédente.

```
pListe AjoutPile(Donnee d, pListe pL) {
  return Constructeur(d, pL);
}
```

// L'ajout dans une file peut se faire récursivement

```
pListe AjoutFile(Donnee d, pListe pL) {
```

```
// Si la file est vide on crée le nouvel élément

  if (pL == NULL) {
    return Constructeur(d, NULL);
  }

// Sinon, on modifie la queue de la file pour insérer le nouvel élément

  Queue(pL) = AjoutFile(d,Queue(pL));

// Et on retourne le maillon courant qui demeure inchangé.

  return pL;
}

// Pour pouvoir libérer la mémoire, il nous faut une fonction de libération

void LibererListe(pListe pL) {

// Toujours de façon récursive, si la liste est vide, il n'y a rien à faire.

  if(pL == NULL)
    return;

// Sinon, on libère d'abord la queue de l'élément courant.

  LibererListe(Queue(pL));

// puis le maillon de l'élément courant.

  free(pL);
}
```

// L'impression se fait de même de façon récursive

```
void Imprime(pListe pL) {
  if(pL ==NULL) {
    printf("\n");
    return;
  }
  printf("[%d]->",Valeur(pL));
  Imprime(Queue(pL));
}
```

// Pour imprimer dans l'autre sens il suffit d'inverser les deux lignes de la récurrence

```
void __ImprimeReverse(pListe pL) {
  if(pL == NULL)
    return;
  __ImprimeReverse(Queue(pL));
  printf("<-[%d]",Valeur(pL));
}
```

// Pour que cela soit plus joli, il faut afficher le caractère de retour chariot en fin d'opération (ce qui doit donc être sorti de la récurrence)

```
void ImprimeReverse(pListe pL) {
  __ImprimeReverse(pL);
  printf("\n");
}
```

// **Tri par insertion**

// La fonction d'insertion suppose la liste `pQ` *triée. Elle insère le maillon pointé par* `pL` *au bon endroit dans la liste.*

```
pListe Insertion(pListe pL, pListe pQ) {
```

// De façon récursive, si la liste pQ *est vide alors le résultat de l'insertion est* pL.

```
  if(pQ == NULL) {
    Queue(pL)=NULL;
    return(pL);
  }
```

// Sinon, comme la liste est triée, alors son plus petit élément est le premier Valeur(pQ). *S'il est plus grand que* Valeur(pL) *alors l'élément* pL *s'insère ici et la queue de* pL *est donc* pQ.

```
  if(Valeur(pQ) >= Valeur(pL)) {
    Queue(pL)=pQ;
    return(pL);
  } else {
```

// Si ce n'est pas le cas, le plus petit élément est bien pQ *et l'élément* pL *soit être inséré dans sa queue. L'opération d'affectation sert à conserver le chaînage de la liste si l'insertion intervient juste après l'élément courant.*

```
    Queue(pQ)=Insertion(pL, Queue(pQ));
    return(pQ);
  }
}
```

// Une fois cette fonction d'insertion rédigée, il suffit de passer en revue tous les éléments de la liste successivement pour la trier.

```
pListe TriInsertion(pListe pL) {
```

// De façon récursive si la liste est vide, il n'y a rien à faire

```
  if(pL == NULL)
    return(pL);
```

// Sinon, on trie la queue de l'élément courant et on insère ensuite celui-ci dans la liste ainsi triée.

```
  Queue(pL) = TriInsertion(Queue(pL));
  pL=Insertion(pL, Queue(pL));
  return(pL);
}
```

// **Tri par fusion**

// Pour ce premier algorithme nous utilisons une fonction qui compte le nombre d'éléments de la liste.

```
int Count(pListe pL) {
```

// Récursivement si la liste est vide elle comporte 0 éléments.

```
  if(pL == NULL)
    return(0);
```

// Sinon il y a un élément plus le nombre contenu dans la queue de l'élément courant.

```
  return(1+Count(Queue(pL)));
}
```

// Nous utilisons ensuite une fonction qui sépare une liste pL *en 2. Plus exactement, si la liste pointée par* pL *comporte plus de* split *éléments (avec* split > 0*) elle est coupée au niveau du* split*-ième élément et la fonction retourne alors la queue de cet élément. Sinon, la liste demeure inchagée et la fonction retourne* NULL.

```
pListe Separation(pListe pL, int split) {
  pListe pret;
```

// De façon récursive, si la liste est vide, il n'y a rien à faire.

```
  if(pL==NULL)
    return(NULL);
```

// Sinon, l'élément courant compte pour un élément. S'il faut couper au niveau de ce premier élément alors on retourne la queue de l'élément courant et on coupe le chaînage.

```
  if(split <= 1) {
    pret=Queue(pL);
    Queue(pL)=NULL;
    return(pret);
  }
```

// S'il ne faut pas couper au niveau du premier élément alors il faut séparer la liste plus loin. On appelle donc récursivement la fonction sur la queue de l'élément courant avec un nombre d'élément décrémenté de 1.

```
  pret=Separation(Queue(pL),split-1);
  return(pret);
}
```

// L'opération de fusion consiste à obtenir à partir de deux listes supposées triées `pLg` *et* `pLd` *une liste triée de tous les éléments.*

```
pListe Fusion(pListe pLg, pListe pLd) {
  pListe pret;
```

// De façon récursive, si l'une des listes est vide, le résultat de la fusion est simplement l'autre liste.

```
  if(pLg == NULL)
    return(pLd);
  if(pLd == NULL)
    return(pLg);
```

// Si les deux listes comportent des éléments, alors les premiers éléments sont forcément les deux plus petits de chaque liste (on trie de façon croissante). Donc on compare ces deux éléments pour trouver le plus petit des deux. C'est ce dernier qui sera retourné.

```
  if(Valeur(pLg) <= Valeur(pLd)) {
    pret = pLg;
```

// C'est la valeur de gauche qui est prise donc il ne reste à traiter que la queue de la liste de gauche.

```
    pLg = Queue(pLg);
  } else {
    pret = pLd;
    pLd = Queue(pLd);
  }
```

// Pour affecter la queue de l'élément à retourner, il suffit de fusionner les valeurs restantes des listes gauches et droites.

```
  Queue(pret)=Fusion(pLg,pLd);
  return(pret);
}
```

// La fonction de tri par fusion proprement dite prend en paramètre le nombre d'éléments de la liste. Comme elle ne sera appelée que par la fonction ci-dessous on peut la déclarer statique, c'est-à-dire locale à ce fichier.

```
static pListe __TriFusion(pListe pL, int taille) {
  pListe pLg, pLd;
  int split;
```

// Si la liste compte moins d'un élément elle est déjà triée. La deuxième partie du test n'est pas effectuée si la fonction est correctement appelée avec le bon nombre d'éléments. Elle n'est là que pour s'assurer qu'il n'y aura pas de débordement en cas d'erreur d'appel.

```
  if((taille <= 1)||(pL == NULL))
    return(pL);
```

// On divise la taille en deux et on sépare la liste avec la fonction ci-dessus.

```
  split=taille/2;
  pLg=pL;
  pLd=Separation(pL,split);
```

// De façon récursive, on trie séparément chaque sous-liste.

```
  pLg=__TriFusion(pLg, split);
  pLd=__TriFusion(pLd, taille-split);
```

// Et on fusionne les deux listes une fois triées pour obtenir notre résultat.

```
  return(Fusion(pLg,pLd));
}
```

// La fonction permettant d'appeler le tri par fusion s'assure juste du bon calcul du nombre d'éléments.

```
pListe TriFusion(pListe pL) {
  int Taille;
  Taille=Count(pL);
  return(__TriFusion(pL,Taille));
}
```

// Pour pouvoir comparer les temps de calculs, nous devons pouvoir recopier une liste avant son tri.

```
pListe Copie(pListe pL) {
  pListe pret;
```

// Toujours de façon récursive, si la liste est vide, il n'y a rien à faire.

```
  if(pL == NULL)
    return(NULL);
```

// Sinon, on recopie d'abord la queue de l'élément courant.

```
  pret=Copie(Queue(pL));
```

// Et le résultat à retourner est obtenu en insérant la valeur de l'élément courant en tête de la copie obtenue.

```
  pret=AjoutPile(Valeur(pL),pret);
  return(pret);
}
```

// Nous déclarons une fonction qui retourne le temps utilisé par le programme depuis son lancement en millisecondes.

```
double runtime(void);
```

// La fonction principale du programme est juste une fonction de test.

```
int main(void) {
  int i;
  pListe pL=NULL, pL2=NULL;
```

// Quelques valeurs insérées "à la main" pour obtenir la chaîne 0,1,2,3,4,5.

```
  pL=AjoutPile(2,pL);
  pL=AjoutFile(3,pL);
  pL=AjoutPile(1,pL);
  pL=AjoutPile(0,pL);
  pL=AjoutFile(4,pL);
  pL=AjoutFile(5,pL);
```

// On vérifie le bon fonctionnement par l'impression

```
  Imprime(pL);
  ImprimeReverse(pL);
```

// On ajoute ensuite un peu plus d'éléments.

```
  for(i=0; i<1000; i++) {
```

// La fonction `lrand48` *retourne un entier pseudo-aléatoire. Ceci permet d'avoir une liste d'éléments un peu plus longue.*

```
    pL=AjoutPile(lrand48()%100000,pL);
  }
```

// On recopie l'état actuel de cette liste avant tri.

```
  pL2=Copie(pL);
```

// On vérifie le tri par insertion

```
  pL=TriInsertion(pL);
  Imprime(pL);
```

// On libère la liste pour reprendre notre copie

```
  LibererListe(pL);
  pL=Copie(pL2);
```

// On vérifie le tri par fusion

```
  pL=TriFusion(pL);
  Imprime(pL);
```

// On libère la liste pour reprendre notre copie. Cette fois-ci sans conserver la copie initiale car nous allons insére à nouveau des éléments en nombre significatif.

```
  LibererListe(pL);
  pL=pL2;
  for(i=0; i<25000; i++)
    pL=AjoutPile(lrand48()%100000,pL);
```

// On recopie l'état actuel de cette liste avant tri.

```
  pL2=Copie(pL);
```

// L'objet du test est maintenant de comparer les temps de calcul. Pour cela on utilise la fonction runtime *définie ci-dessous.*

```
  {
```

// Nous déclarons un sous-bloc avec des variables locales (juste pour voir que c'est possible ;-)

```
    double topdepart,toparrivee;
    int count;
```

// Nous comptons les éléments pour notre affichage ci-dessous.

```
    count=Count(pL);
    topdepart=runtime();
    pL=TriInsertion(pL);
    toparrivee=runtime();
    fprintf(stderr,"Tri par insertion : "
    "liste de %d éléments triée en %.2f s\n",
    count,(toparrivee-topdepart)/1000.0);
```

// Exemple de temps d'exécution pour 26006 éléments avec l'option -g *18.47 secondes, avec l'option d'optimisation du compilateur* -O3 *15.31 secondes.*

```
    topdepart=runtime();
    pL2=TriFusion(pL2);
    toparrivee=runtime();
    fprintf(stderr,"Tri par fusion : "
    "liste de %d éléments triée en %.2f s\n",
    count,(toparrivee-topdepart)/1000.0);
```

// Exemple de temps d'exécution pour 26006 éléments avec l'option -g *0.03 seconde, avec l'option d'optimisation du compilateur* -O3 *0.02 seconde. Le gain algorithmique est donc beaucoup plus intéressant que celui obtenu en optimisant un code. On pourra par exemple implanter une fonction de séparation de liste qui n'a pas besoin de compter les éléments de la liste en les prenant un sur deux. Cette approche pourrait paraître plus rapide mais on constatera alors que les réarrangements de pointeurs nécessaires sont plus coûteux que de compter le nombre d'éléments.*

```
  }
  LibererListe(pL);
  LibererListe(pL2);
  return(0);
}
```

// La fonction runtime *utilise des appels systèmes. Voir le manuel de* times(2) *pour plus de précision.*

```
#include <time.h>
#include <limits.h>
#include <sys/time.h>
#include <sys/times.h>
#include <sys/param.h>
#ifndef CLK_TCK
# define CLK_TCK sysconf(_SC_CLK_TCK)
#endif
#ifndef HZ
# define HZ CLK_TCK
#endif
double runtime (void)
{
  struct tms t;
  double ret;
  times(&t);
  ret=t.tms_utime*((double)1000.)/HZ;
  return(ret);
}
```

// Fonction de séparation sans comptage des éléments de la liste... moins efficace ! ! !

```
pListe Separation2(pListe pL) {
  pListe pret;
```

// De façon récursive, si la liste est vide, il n'y a rien à faire.

```
  if(pL==NULL)
    return(NULL);
```

// Sinon, l'élément courant reste inchangé et sa queue va dans la liste retournée.

```
  pret=Queue(pL);
  if(pret == NULL)
    return(pret);
```

// Si le pointeur retourné n'est pas une liste vide, alors on réarrange la liste en sautant l'élément.

```
  Queue(pL)=Queue(pret);
```

// Et on rappelle la fonction sur la queue.

```
  Queue(pret)=Separation2(Queue(pret));
  return(pret);
}
```

D Commandes de base Unix

Le tableau ci-dessous décrit un certain nombre de fonctions de base du système **Unix**. La colonne “Type” indique par un “P” les programmes et par un “X” un programme tournant uniquement sous **X**. Les commandes internes au shell sont signalées par “S” pour les commandes du shell **bash** (ou **sh**) et par “C” pour les commandes du shell **csh**.

Commande	Type	Syntaxe	Description
alias	C	**alias nom_alias ”commande”**	En **csh**, permet de définir un raccourci de commande. En outre, \ **!*** peut être utilisé dans le libellé pour remplacer ses arguments. Il est donc possible de réaliser ainsi de véritables commandes.
alias	S	**alias nom_alias = ”commande”**	Syntaxe légèrement différente en **bash**.
ar	P	**ar -crs [nom de fichier .a] [fichiers .o]**	Ce programme constitue des archives de fichiers. Il est principalement utilisé pour créer des fichiers de bibliothèque statique avec les options : c créer le fichier s'il n'existe pas ; r remplacer les fichiers existants éventuellement dans l'archive par ceux spécifiés sur la ligne de commande ; s fabriquer un index des fonctions contenues dans les fichiers binaires .o pour faciliter l'édition de liens.

Commande	Type	Syntaxe	Description
awk	P	**awk -Fc 'programme' fichier** **awk -f programme fichier**	Ce programme permet de gérer des fichiers ligne par ligne et/ou de sélectionner des colonnes. Le séparateur **c** des colonnes est indiqué par l'option **-F**. Le programme de sélection peut être indiqué en argument ou placé dans un fichier de programmation désigné par l'option **-f**. La syntaxe de ces programmes est celle du C, avec quelques possibilités supplémentaires au niveau des chaînes de caractères.
bash	P	**bash**	Le Bourne shell évolué.
bg	CS	**bg**	Permet de relancer en tâche de fond un programme suspendu par Ctrl-Z.
cat	P	**cat** **cat file**	Selon la syntaxe, recopie l'entrée standard ou le fichier **file** sur la sortie standard.
cc	P	**cc**	Le compilateur C par défaut du système.
chmod	P	**chmod u+rx fichiers** **chmod g-w fichiers**	Permet de changer les droits d'accès à un fichier. La syntaxe comporte l'une des lettres suivantes o *owner* le possesseur du fichier. g groupe. u utilisateur. suivie de l'action à effectuer + ajoute le droit. - retranche le droit. = donne le droit. suivie d'une combinaison des caractères r *read* droit en lecture. w *write* droit en écriture. x *eXecute* droit en exécution. qui s'applique aux fichiers.
cp	P	**cp file1 file2**	Copie le fichier **file1** dans le fichier
csh	P	**csh**	Le C-shell de base.

Commande	Type	Syntaxe	Description
cut	P	**cut -f1** **cut -f1 -d\(**	Permet de tronçonner une entrée en champs successifs par **-f1**, **-f2**, **-f3**, etc... Le séparateur de champ peut être précisé par l'option **-d**.
echo	CS	**echo "Hello world"**	**echo** permet de faire écrire au shell son argument.
egrep	P	**egrep -e'motif' fichiers**	Cette commande permet de rechercher dans un fichier un motif particulier. Le motif utilise une syntaxe particulière, par exemple : "*" désigne n'importe quelle chaîne. "." désigne n'importe quelle caractère. "^" désigne le début d'une ligne.
emacs	P(X)	**emacs fichier**	Cet éditeur de fichiers est particulièrement efficace sous **X**. Il offre en effet une approche facilitée par l'intermédiaire de menus déroulants, tout en conservant des raccourcis claviers rapides. On notera par exemple : – "Ctrl-X-Ctrl-F" pour ouvrir un fichier. – "Ctrl-X-Ctrl-C" pour quitter **emacs**. – "Ctrl-X-Ctrl-S" pour sauvegarder un fichier. Il reconnaît en outre un grand nombre de types de fichiers et facilite l'édition de ces fichiers en adaptant son comportement au langage du fichier. Ainsi, les dernières versions d'emacs assurent elles des indentations et des colorations adaptées pour les langages C, les shells-scripts, **LaTeX**, etc...
env	P	**env**	Cette commande affiche toutes les variables d'environnement et leurs valeurs.

Commande	Type	Syntaxe	Description
exec	CS	**exec cmd**	Exécute une commande de manière définitive, le shell ne reprenant pas la main à l'issue. Le code de sortie du shell est alors celui de la commande exécutée.
exit	CS	**exit 0**	Arrête le shell avec le code de sortie indiqué.
export	S	**export VAR=valeur**	Spécifie la variable **VAR** comme variable d'environnement à passer à tous les processus fils du shell.
fg	CS	**fg**	Permet de relancer en tâche de fond un programme suspendu par Ctrl-Z.
find	P	**find**	Permet de rechercher récursivement dans un répertoire des fichiers qui satisfont des critères et de leur appliquer des commandes (voir le manuel).
for	S	**for var in (liste) ; do (cmd) ; done ;**	Commande de boucle de **sh**.
ftp	P	**ftp**	Programme de transfert de fichiers à distance.
function	S	**function**	En **bash**, permet de définir des fonctions nouvelles.
gcc	P	**gcc**	Compilateur C de **Gnu** particulièrement adapté au débugger **gdb**.
gdb	P	**gdb**	Debugger de programme C particulièrement bien adapté à **gcc** et **emacs**.
if	S	**if [test] ; then (cmd) ; else (cmd) ; fi ;**	Commande de branchement conditionnel de **sh**.
jobs	CS	**jobs**	Commande permettant de vérifier les statuts de tous les processus du shell lancés en tâche de fond.
kill	CS	**kill PID**	Commande permettant d'envoyer des signaux à un processus.

Commande	Type	Syntaxe	Description
less	P	**less fichier**	Programme facilitant la lecture d'un fichier long.
ln	P	**ln file1 file2** **ln -s file1 file2**	Commande donnant au fichier **file2** le même contenu physique que **file1**. L'option **-s** permet d'effectuer un lien symbolique ; ceci permet en particulier de relier des fichiers ne se trouvant pas sur le même disque.
ls	P	**ls -al**	Programme listant les fichiers d'un répertoire. L'option **-a** permet de lister aussi les fichiers commençant par un caractère ".", tandis que l'option **-l** utilise un format long.
make	P	**make**	Utilitaire de compilation séparée de fichiers.
man	P	**man man** **man csh** **man -k users**	**man** est un manuel en ligne (et en anglais...) des commandes d'**Unix**, mais contient aussi toutes les descriptions des bibliothèques de fonctions C du système. Lorsqu'il est correctement installé (!) **man** permet aussi de rechercher les rubriques associées à un concept grâce à l'option **-k**. La plupart des pages de manuel se terminent par une rubrique *SEE ALSO* qui contient des références à d'autres pages du manuel. TOUT est dans le manuel, mais trouver une information peut parfois passer par des chemins détournés !
mkdir	P	**mkdir DIR**	Programme de création de répertoire.
more	P	**more fichier**	Programme facilitant la lecture d'un fichier long, moins performant que **less**, mais toujours présent.
mv	P	**mv file1 file2** **mv -i file1 file2**	Programme qui renomme le fichier **file1** en **file2**. L'option **-i** provoque une demande de confirmation si un fichier est sur le point d'être écrasé par l'opération.

Commande	Type	Syntaxe	Description
ps	P	**ps**	Programme permettant de visualiser les numéros des processus.
rm	P	**rm *.o** **rm -ir tmp**	Programme permettant d'effacer des fichiers. L'option **-i** permet d'assurer un fonctionnement interactif plus sûr (chaque effacement demande confirmation). L'option **-r** effectue un effacement des répertoires, par effacement récursif de tous les fichiers qu'il contient.
rmdir	P	**rmdir DIR**	Efface un répertoire vide.
setenv	C	**setenv VAR valeur**	En **csh**, permet d'affecter à la variable **VAR** la valeur **test** et le statut de variable d'environnement passée à tous les processus fils du shell.
sh	P	**sh**	Le Bourne-shell de base.
sleep	P	**sleep**	**sleep** attend le nombre de secondes indiqué en argument.
test	P	**test "a" = "b"** **test -f file**	Programme de test adapté à la commande **if** qui permet les tests usuels sur les chaînes de caractères ou les nombres. Il permet aussi de tester l'existence et l'accessibilité de fichiers.
touch	P	**touch file**	programme permettant de simplement modifier la date de dernière modification d'un fichier.
uname	P	**uname** **uname -a**	Programme décrivant le système **Unix**utilisé. L'option **-a** permet d'obtenir plus de renseignements sur la machine.
vi	P	**vi file**	Éditeur de fichiers très peu intuitif, mais standard en **Unix**.
who	P	**who**	Programme indiquant les personnes connectées sur une machine.
xman	X	**xman**	**xman** est un utilitaire qui interface dans une fenêtre **X** la commande **man**.

Commande	Type	Syntaxe	Description
xterm	S	**xterm** **xterm -e cmd**	Programme **X**, permettant d'ouvrir un terminal sur la machine dans une fenêtre. L'option **-e** ouvre une fenêtre et y exécute la commande **cmd**.

E Index

alias, 91
ar, 51, 91
awk, 92

bash, 91, 92, 94
bg, 92
bloc , 9, 12–15, 17, 18
bord (effet de), 24

langage C, 7, 9–11, 13, 14, 16, 18, 19, 23, 25–27, 29, 35, 41–43, 51, 67, 92–95
cat, 92
cc, 92
chmod, 92
cp, 92
csh, 91, 92, 96
cut, 93

echo, 24, 93
effet de bord, 24
egrep, 93
emacs, 32–34, 38, 39, 67, 93, 94
env, 93
exec, 94
exit, 94
export, 94

fg, 94
find, 94
for, 94
ftp, 94
function, 94

gcc, 32, 94
gdb, 32–35, 38, 39, 67, 94
Gnu, 94

if, 94, 96

jobs, 94

kill, 94

LaTeX, 93
less, 95
ln, 95
ls, 95

main, 19, 33
make, 95
man, 95, 96
mkdir, 95
more, 95
mv, 95

ps, 96

rm, 96
rmdir, 96

setenv, 96
sh, 91, 94, 96
sleep, 96

test, 96
touch, 96

uname, 96
système **Unix**, 19, 29, 31, 32, 51, 91, 95, 96

vi, 96

who, 96

serveur **X**, 91, 93, 96, 97
xman, 96
xterm, 97