Méthodologie de la programmation en langage C

Principes et applications

J.-P. BRAQUELAIRE

2e édition

MASSON

Pour la liste complète, voir catalogue

MÉTHODES DE PROGRAMMATION ET ALGORITHMIQUE

ANALYSE INFORMATIQUE. P.-A. Goupille.
CONCEPTION OBJET DES STRUCTURES DE DONNÉES. B. Quément.
ÉLÉMENTS D'ALGORITHMIQUE. D. Beauquier, J. Berstel et Ph. Chrétienne.
CONCEPTION ET PROGRAMMATION PAR OBJETS. Techniques, outils, et applications. J.-P. Aubert et P. Dixneuf.
• ALGORITHMIQUE. Conception et analyse. G. Brassard et P. Bratley.
PROGRAMMATION IMPÉRATIVE ET PROGRAMMATION DÉCLARATIVE. Ph. Collard.
• INTRODUCTION À LA PROGRAMMATION
1. Algorithmique et langages. J. Biondi et G. Clavel.
2. Structures de données G. Clavel et J. Biondi.
3. Exercices corrigés G. Clavel et F.B. Jorgensen.
• SCHÉMAS ALGORITHMIQUES FONDAMENTAUX P.C. Scholl et J.P. Peyrin.
ASSEMBLAGE, MODÉLISATION, PROGRAMMATION (80x86). M. Margenstern.
LA COMPRESSION DES DONNÉES. Méthodes et applications. G. Held.
• ALGORITHMIQUE ET PRÉSENTATION DES DONNÉES
1. Files, automates d'états finis. M. Lucas, J.-P. Peyrin et P.-C. Scholl.
2. Évaluations, arbres, graphes, analyse de texte. M. Lucas.
3. Récursivité et arbres. P.-C. Scholl.
PROCESSUS CONCURRENTS. Introduction à la programmation parallèle. M. Ben Ari.
PROCESSUS SÉQUENTIELS COMMUNICANTS. C.A.R. Hoare.
CONSTRUCTION ET VÉRIFICATION DE PROGRAMMES. R. Backhouse.

LES LANGAGES ET LEUR TRAITEMENT

LE LANGAGE C. B.W. Kernighan et D.M. Ritchie.
LE LANGAGE C. Solutions. C.L. Tondo et S.E. Gimpel.
CONSTRUCTION LOGIQUE DE PROGRAMMES COBOL. Mise à jour COBOL 85. M. Koutchouk.
• LANGAGE C norme ANSI. Vers une approche orientée objet. Ph. Drix.
TURBO INITIATION À LA PROGRAMMATION EN PASCAL, pour Turbo-Pascal 4.0, 5.0, 5.5, 6.0. J. Thiel, C. Léger et G. Jacquet.
• MÉTHODOLOGIE DE LA PROGRAMMATION EN LANGAGE C. Principes et applications. J.-P. Braquelaire.
• (COMMON) LISP. Une introduction à la programmation. H. Wertz.
COBOL. Perfectionnement et pratique. M. Koutchouk.
PROGRAMMER EN C++. S.C. Dewhurst et K.T. Stark.
LE GÉNÉRATEUR AUTOMATIQUE DE PROGRAMME RPG. M. Rémy.
LANGAGE C : MANUEL DE RÉFÉRENCE. S.H. Harbison et G.L. Steele.
LANGAGE C. PROBLÈMES ET EXERCICES. A.R. Feuer.
• LANGAGE C, norme ANSI. Variations sur des thèmes Pascal. Ph. Drix.
• LES LANGAGES DE PROGRAMMATION. Concepts essentiels, évolution et classification. J. Lonchamp.
INTRODUCTION AU LANGAGE ADA. D. Price.
• TRAITEMENT DES LANGAGES ÉVOLUÉS. Compilation. Interprétation. Support d'exécution. Y. Noyelle.
• APPRENDRE PASCAL ET LA RÉCURSIVITÉ. Avec exemples en TURBO PASCAL. R. Romanetti.
LE LANGAGE PASCAL. J.-M. Crozet et D. Serain.
MANUEL ADA. LANGAGE NORMALISÉ COMPLET. M. Thorin.

INFORMATIQUE THÉORIQUE

THÉORIE DES LANGAGES ET DES AUTOMATES. J.-M. Autebert.
CALCULABILITÉ ET DÉCIDABILITÉ. J.-M. Autebert.

• Cours rédigé et enseigné par un professeur francophone.

(Suite page 3 de couverture

Méthodologie de la programmation en langage C

CHEZ LE MÊME ÉDITEUR

Dans la collection MIM

LE LANGAGE C, NORME ANSI, par B.W. Kernighan et D.M. Ritchie. Traduit de l'anglais par J.F. Groff et E. Mottier. 1994, 2e édition, 4e tirage, 296 pages.

LE LANGAGE C, solutions, aux exercices de la 2e édition de l'ouvrage de B.W. Kernighan et D.M. Ritchie, par C.L. Tondo et S.E. Gimpel. Traduit de l'anglais par D. Bertier. 1992, 2e édition, 2e tirage, 168 pages.

LANGAGE C NORME ANSI, vers une approche orientée objet, par Ph. Drix. 1993, 2e édition revue et augmentée, 440 pages.

CONCEPTION OBJET DES STRUCTURES DE DONNÉES. Réalisation en langage C, par B. Quément. 1992, 248 pages.

LANGAGE C. Problèmes et exercices, par A.R. Feuer. Traduit de l'anglais par C. Arthur et J. Kott. 1991, 2e édition, 184 pages.

LANGAGE C : MANUEL DE RÉFÉRENCE, par S.P. Harbison et G.L. Jr Steele. Traduit de l'anglais par J.Cl. Franchitti. 1990, 472 pages.

LE DÉVELOPPEMENT DE LOGICIEL EN C++, par R. Winder. Traduit de la 2e édition anglaise par P.-Y. Bonnetain. 1994, 568 pages.

COMPRENDRE ET UTILISER C++ POUR PROGRAMMER OBJET, par G. Clavel, I. Trillaud, L. Veillon. 1994, 248 pages.

PROGRAMMER EN C++, par S.C. Dewhurst et K.T. Stark. Traduit de l'anglais par J.-F. Groff. 1990, 208 pages.

LE PASCAL ET LE LANGAGE C APPLIQUÉS À LA ROBOTIQUE, par Ch. Blume, W. Jakob, J. Favaro. Traduction de Ch. Ricard et Y. Arrouye. 1988, 288 pages.

Autres ouvrages

C++, par B. Beaudoing et D. Edelson. Préface de Ph. Gautron. *Collection Objectif...* 1994, 196 pages.

C, C++ ET UNIX. Initiation aux langages et environnement, avec exercices, par G. Khalil. *Collection Techniques de l'Informatique*. 1991, 176 pages.

PROGRAMMER EN C, par M.I. Bolsky et AT&T. Traduit de l'anglais par M. Calleux. *Collection des Mémo-Guides*. 1987, 88 pages.

MANUELS INFORMATIQUES MASSON

Méthodologie de la programmation en langage C

Principes et applications

Jean-Pierre BRAQUELAIRE

Professeur d'informatique
Université Bordeaux I

2e édition

2e tirage

MASSON
Paris Milan Barcelone
1995

Ce logo a pour objet d'alerter le lecteur sur la menace que représente pour l'avenir de l'écrit, tout particulièrement dans le domaine universitaire, le développement massif du « photocopillage ».
Cette pratique qui s'est généralisée, notamment dans les établissements d'enseignement, provoque une baisse brutale des achats de livres, au point que la possibilité même pour les auteurs de créer des œuvres nouvelles et de les faire éditer correctement est aujourd'hui menacée.
Nous rappelons donc que la reproduction et la vente sans autorisation, ainsi que le recel, sont passibles de poursuites. Les demandes d'autorisation de photocopier doivent être adressées à l'éditeur ou au Centre français d'exploitation du droit de copie : 3, rue Hautefeuille, 75006 Paris. Tél. : 43 26 95 35.

En couverture : Vue d'artiste des satellites Spot 4, Télécom 2 et du lanceur Ariane 5.

Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit, des pages publiées dans le présent ouvrage, faite sans l'autorisation de l'éditeur, est illicite et constitue une contrefaçon. Seules sont autorisées, d'une part, les reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective, et d'autre part, les courtes citations justifiées par le caractère scientifique ou d'information de l'œuvre dans laquelle elles sont incorporées (art. L. 122-4, L. 122-5 et L. 335-2 du Code de la propriété intellectuelle).

Des photocopies payantes peuvent être réalisées avec l'accord de l'éditeur. S'adresser au : Centre français d'exploitation du droit de copie, 3, rue Hautefeuille, 75006 Paris, tél.: 43.26.95.35.

© *Masson, Paris, 1990, 1993*

ISBN : 2-225-**84353**-8

ISSN : 0249-6992

MASSON S.A. 120, bd Saint-Germain, 75280 Paris Cedex 06
MASSON S.p.A. Via Statuto 2/4, 20121 Milano
MASSON S.A. Avenida Principe de Asturias 20, 08012 Barcelona

Table des matières

Avant-propos **xv**

Introduction **1**

I Structure du langage C 5

1 Introduction à la programmation en langage C **7**
- 1.1 Quelques fonctions et c'est tout un programme 7
- 1.2 Du programme source au fichier exécutable 9
  - 1.2.1 Compilation et édition de liens 9
  - 1.2.2 Commande `make` et fichier `Makefile` 13
- 1.3 Type et valeur des expressions 16
- 1.4 Déclaration des fonctions 17
  - 1.4.1 Type d'une fonction 17
  - 1.4.2 Prototype d'une fonction 19
- 1.5 La fonction `printf` 21
- 1.6 La fonction `main` 23
- 1.7 Variables et effets de bord 26
  - 1.7.1 Déclaration et initialisation de variables 26
  - 1.7.2 Affectation de variables 27
  - 1.7.3 Paramètres passés par référence 27
  - 1.7.4 Variables globales 27
- 1.8 Tests et itérations 30
  - 1.8.1 Test : `if-else` 30
  - 1.8.2 Itération : `while` 33
  - 1.8.3 Itération : `for` 36
- 1.9 Quelques règles de présentation 38
  - 1.9.1 Découpage en lignes 39
  - 1.9.2 Indentation 41
- 1.10 GNU Emacs et le langage C 43
  - 1.10.1 Marquage de régions et indentation 43
  - 1.10.2 Compilation et correction d'erreurs 44

2 Les déclarations **47**
- 2.1 Types de base et types élémentaires 47
  - 2.1.1 Types de base 47
  - 2.1.2 Types élémentaires 50
  - 2.1.3 Types prédéfinis 52
- 2.2 Mécanismes de construction d'objets 53
- 2.3 Constructeurs homogènes 55
- 2.4 Constructeurs hétérogènes 58

2.4.1 Regroupement d'objets : les structures 58
2.4.2 Champs de bits 59
2.4.3 Unions d'objets 60
2.5 Définition de types 62
2.5.1 Le constructeur typedef 62
2.5.2 Expressions de type 64
2.6 Implémentation et visibilité 65
2.6.1 Niveau d'une variable 65
2.6.2 Implémentation des variables 66
2.6.3 Optimisation du code généré 69
2.6.4 Variables volatiles 72
2.6.5 Règles de visibilité 73
2.6.6 Définitions et références 76
2.7 Initialisation de variables 77
2.7.1 Variables permanentes 77
2.7.2 Variables temporaires 81

3 Les expressions **85**
3.1 Expressions élémentaires 85
3.1.1 Choix des identificateurs 85
3.1.2 Constantes littérales 87
3.1.3 Constantes symboliques 92
3.1.4 Variables et g-valeurs 95
3.2 Mécanismes de construction 96
3.2.1 Constructions d'expressions 96
3.2.2 Priorité des opérateurs 97
3.2.3 Associativité des opérateurs 98
3.2.4 Type des opérandes 99
3.3 Syntaxe et sémantique des opérateurs 100
3.3.1 Expressions primaires 100
3.3.2 Conversion de types 102
3.3.3 Expressions arithmétiques et logiques 103
3.3.4 Arithmétique booléenne et manipulation de bits 106
3.3.5 Manipulation d'adresses 109
3.3.6 Taille d'un objet 111
3.3.7 Listes d'expressions et expressions conditionnelles 112
3.3.8 Opérateurs à effet de bord 114
3.4 A propos des effets de bord 118

4 Les instructions **121**
4.1 Instructions élémentaires 121
4.2 Structures de contrôle 123
4.2.1 Test 123
4.2.2 Aiguillage 125
4.2.3 Itérations 129
4.3 Instructions d'échappement 134
4.3.1 Echappements structurés 135
4.3.2 Branchement inconditionnel 140
4.3.3 Fonctions provoquant une rupture de séquence 141

II Environnement et méthodes de programmation 145

5 L'utilisation du préprocesseur **147**

5.1 Définition de pseudo-fonctions 147
5.1.1 Pseudo-constantes 147
5.1.2 Définition à la compilation 153
5.1.3 Pseudo-constantes prédéfinies 153
5.1.4 Pseudo-fonctions 154
5.1.5 Récursivité des définitions 159
5.2 Inclusion de fichiers 159
5.2.1 La directive include 159
5.2.2 Fichiers en-tête 161
5.2.3 Fichiers en-tête standard 161
5.3 Compilation conditionnelle 163
5.3.1 Test d'existence d'une pseudo-constante 163
5.3.2 Prise en compte de l'implémentation du compilateur 165
5.3.3 Évaluation de pseudo-expressions 167
5.4 Pilotage des messages d'erreur 169
5.5 Un exemple de construction générique 170

6 Manipulation d'adresses **175**
6.1 Expressions et pointeurs 175
6.1.1 Valeur d'un pointeur 175
6.1.2 Affectations de pointeurs et conversions de types 176
6.1.3 Additions et pointeurs 178
6.1.4 Initialisation de pointeurs 179
6.2 Pointeurs et chaînes de caractères 181
6.3 Pointeurs et vecteurs 184
6.3.1 Similitudes et différences 184
6.3.2 Vecteur de pointeurs et pointeur de pointeurs 185
6.4 Pointeurs et fonctions 185
6.4.1 Fonctions retournant un pointeur 185
6.4.2 Passage par référence 187
6.4.3 Fonctions passées en paramètres 187
6.4.4 Tables d'indirection de fonctions 188
6.5 Manipulation de listes 191

7 La bibliothèque standard **195**
7.1 Les entrées-sorties standard 195
7.1.1 Principes généraux 195
7.1.2 Mécanismes de formatage 196
7.1.3 Ouverture et fermeture de flot 202
7.1.4 Fonctions de lecture et d'écriture 205
7.1.5 Accès séquentiel et accès direct 208
7.1.6 Contrôle de la mémorisation 209
7.2 Manipulation de caractères et de chaînes 211
7.2.1 Type d'un caractère 211
7.2.2 Valeur d'une chaîne numérique 213
7.2.3 Chaînes de caractères 214
7.3 Allocation dynamique de mémoire 221
7.4 Assertions et mise au point de programme 229
7.5 Accès à l'environnement utilisateur 234
7.6 Temps interne et temps externe 236
7.7 Fonctions avec paramètres variables 240
7.8 Lancement d'une commande 246
7.9 Sauvegarde et restauration du contexte 248
7.9.1 Contexte d'exécution 248

7.9.2 Les fonctions setjmp et longjmp 249
7.9.3 Limites du mécanisme de sauvegarde/restauration 252

8 Modularisation des programmes C **255**
8.1 Quelques principes de modularité 255
8.1.1 Définition des objectifs 255
8.1.2 Principes de base 256
8.2 La fonction comme unité modulaire 259
8.2.1 Fonctions pures 259
8.2.2 Fonctions avec environnement 260
8.2.3 Gestion de plusieurs fonctionnalités 265
8.2.4 Limites de l'implémentation de modules par des fonctions . . 270
8.3 Le fichier comme unité modulaire 270
8.3.1 Mise en œuvre 270
8.3.2 Un exemple plus complet 273
8.3.3 Limites de l'implémentation de modules par des fichiers . . . 281
8.4 Un modèle de programmation par objets 281
8.4.1 Quelques aspects de la programmation par objets 281
8.4.2 Une implémentation des classes en C 283
8.4.3 Gestion de la mémoire 291
8.5 Paramétrer pour réutiliser 298
8.6 Une alternative à l'itération 306
8.7 Objets génériques 313
8.7.1 Modules génériques ou objets génériques 313
8.7.2 La classe "vecteur générique extensible" 314
8.7.3 La classe "pile générique" 318
8.8 Vers le polymorphisme 321

III Interface avec le système Unix 323

9 Les appels système **325**
9.1 Interface avec le noyau UNIX 325
9.2 Récupération des erreurs 326
9.3 Identification des ressources 328

10 Entrées-sorties de bas niveau **329**
10.1 Organisation des entrées-sorties 329
10.1.1 Pilotes d'entrée-sortie 329
10.1.2 Descripteurs de fichier 331
10.2 Mécanismes de base 332
10.2.1 Ouverture et fermeture 332
10.2.2 Lecture et écriture 338
10.2.3 Positionnement en accès direct 346
10.3 Contrôle des entrées-sorties 349
10.3.1 Redirection des entrées-sorties standard 349
10.3.2 Détection de fin de fichier 351
10.3.3 Lectures non bloquante 352
10.3.4 Interactions entre haut niveau et bas niveau 355
10.4 Configuration des pilotes de terminaux 356
10.4.1 Fonctionnement d'un pilote tty 356
10.4.2 Mise en œuvre sous Bsd 359
10.4.3 Mise en œuvre sous System V 361
10.4.4 Un exemple d'utilisation 364

11 Manipulation des fichiers **369**
11.1 Organisation du système de fichiers Unix 369
11.1.1 Bloc de description de fichiers : inode 369
11.1.2 Attributs d'un fichier 370
11.1.3 Organisation des répertoires 378
11.1.4 Modification des attributs d'un fichier 384
11.2 Création et suppression de fichiers 385
11.2.1 Masque de création de fichiers 385
11.2.2 Verrouillage de ressources 386
11.2.3 Créations et suppressions de liens 389
11.2.4 Répertoires et fichiers spéciaux 396
11.3 Gestion de fichiers indexés 399
11.4 Utilisation de fichiers système 405

12 Manipulation des processus Unix **409**
12.1 Structure d'un processus 409
12.1.1 Contexte d'un processus 409
12.1.2 Structure d'un programme exécutable 415
12.1.3 Régions d'un processus 420
12.1.4 Démarrage et terminaison d'un processus 422
12.1.5 Sauvegarde d'image mémoire 424
12.1.6 Chargement dynamique 429
12.2 Gestion des signaux 433
12.2.1 Signalerie Unix 433
12.2.2 Envoi de signal 434
12.2.3 Récupération de signal 437
12.2.4 Gestion d'une alarme 441
12.3 Création de processus 444
12.3.1 Substitution d'un processus 444
12.3.2 Duplication d'un processus 450
12.3.3 Héritage du contexte système 454

13 Communications interprocessus **459**
13.1 Synchronisation de processus par signaux 459
13.1.1 Attente d'une terminaison 459
13.1.2 Attente d'un signal 461
13.2 Utilisation des tubes standard 465
13.3 Redirection d'entrée-sortie dans un tube 472

A Mémento des définitions standard **479**
A.1 Principales définitions de type 479
A.2 Principales définitions de constantes 480
A.3 Variables globales standard 481
A.4 Fonctions et pseudo-fonctions standard 482

B Fichiers en-tête standard **497**

C Table des codes ASCII **499**

Index des programmes **501**

Index général **505**

Contents

Preface xv

Introduction 1

I Structure of the C Language 5

1 An Overview of C Programming 7
1.1 From Functions to Programs 7
1.2 From Source File to Command 9
1.3 Type and Value of Expressions 16
1.4 Declaration of Functions 17
1.5 The `printf` Function 21
1.6 The `main` Function 23
1.7 Variables and Side Effects 26
1.8 Tests and Loops 30
1.9 Some Rules for Program Presentation 38
1.10 GNU Emacs and the C Language 43

2 Declarations 47
2.1 Basic Types 47
2.2 Object Construction Rules 53
2.3 Homogeneous Constructors 55
2.4 Heterogeneous Constructors 58
2.5 Type Definitions 62
2.6 Extent and Scope of Variables 65
2.7 Initialization of Variables 77

3 Expressions 85
3.1 Elementary Expressions 85
3.2 Construction Rules 96
3.3 Syntax and Semantic of Operators 100
3.4 About Side Effects 118

4 Instructions 121
4.1 Elementary Instructions 121
4.2 Test and Loop Instructions 123
4.3 Jumps 134

II Methods and the Programming Environment 145

5 The C Preprocessor 147
5.1 Definition of Pseudo-functions 147
5.2 File Inclusion 159
5.3 Conditional Inclusion 163
5.4 Error Message Handling 169
5.5 An Example of Generic Construction 170

6 Address and Pointers 175
6.1 Expressions and Pointers 175
6.2 Pointers and Strings 181
6.3 Pointers and Arrays 184
6.4 Pointers and Functions 185
6.5 List Manipulation 191

7 The Standard Library 195
7.1 Standard Input/Output 195
7.2 Character and String Manipulation 211
7.3 Memory Allocation 221
7.4 Assertions and Program Debugging 229
7.5 Access to the Environment 234
7.6 Internal and External Time 236
7.7 Functions with Variable Parameter Lists 240
7.8 Command Execution 246
7.9 Non-local Jumps 248

8 Modular Decomposition of Programs 255
8.1 About Modular Decomposition 255
8.2 Modules as Functions 259
8.3 Modules as Files 270
8.4 A Model of Object Oriented Programming 281
8.5 Data Driven Programming 298
8.6 An Alternative to Loop Instructions 306
8.7 Generic Objects 313
8.8 Towards Polymorphism 321

III Interface with the UNIX System 323

9 The System Call Interface 325
9.1 Interface with the UNIX Kernel 325
9.2 Error Handling 326
9.3 Identification and Access 328

10 Low Level Input/Output 329
10.1 The Input/Output Organization 329
10.2 Basic Mechanisms 332
10.3 Input/Output Control 349
10.4 Configuration of Tty Drivers 356

11 Interaction with the File System 369
11.1 The Organization of the File System 369
11.2 File Creation and Deletion 385

11.3 Indexed Files 399
11.4 About the System Files 405

12 Process Manipulation **409**
12.1 Structure of a Process 409
12.2 Signal Handling 433
12.3 Process Creation 444

13 Inter-Process Communication **459**
13.1 Communication using Signals 459
13.2 Communication by Pipes 465
13.3 Pipe Redirections 472

A Glossary of Standard Definitions **479**

B Standard Headers **497**

C ASCII Codes **499**

Index of Programs **501**

General Index **505**

Liste des tables

2.1 Types de base 48
2.2 Constructeurs d'objets 53

3.1 Caractères spéciaux 90
3.2 Liste des *trigraphs* 90
3.3 Opérateurs d'expression rangés par priorité décroissante 99
3.4 Expressions arithmétiques et logiques 105
3.5 Expressions de manipulation de bits 108
3.6 Expressions de manipulation d'adresse 111
3.7 Expression *taille-de* 111
3.8 Liste d'expressions et expression conditionnelle 113

6.1 Opérateurs additifs et pointeurs 178

7.1 Spécifications de conversion de format 198
7.2 Modes d'ouverture d'un flot d'entrées-sorties 203
7.3 Type d'un caractère : `ctype.h` 212
7.4 Description du temps externe 237
7.5 Formats reconnus par la fonction `strftime` 239

12.1 Principaux signaux UNIX 434
12.2 Description de l'appel `kill`(*pid*, *sig*) 435

Avant-propos

Personne ne sait vraiment définir ce qu'est le *"swing"* et paradoxalement tout le monde sait le reconnaître. Il en va de même avec le style en programmation : personne ne se risquerait à donner la définition d'un programme *"bien écrit"*, mais tout programmeur expérimenté est en général capable d'apprécier la qualité d'un programme à sa seule lecture. Et très souvent, les avis en la matière convergent.

Plus précisément, deux spécialistes du langage C sont en général d'accord pour encenser tel ou tel style de programmation, on dit parfois paradigme, et rejeter tel autre que, bien évidemment, deux experts du langage ADA revendiquent. Par conséquent, lorsqu'on parlera dans ce livre de *programme bien écrit*, ou lorsque l'on donnera un *conseil de programmation*, il faudra comprendre *programme* C *bien écrit*, ou *conseil de programmation en* C.

Une des caractéristiques fondamentales de C est d'assurer la continuité entre le haut et le bas niveau au sein d'un même *formalisme*. Nous pensons que c'est ce qui fait son grand intérêt. Seulement, un langage de programmation ne peut privilégier la puissance d'expression et la souplesse d'utilisation, et être dans un même temps l'incarnation d'un style de programmation particulier. C'est la contrepartie : C est un langage à risques, ne véhiculant aucune méthode spécifique.

Le *"style* C*"* s'est développé peu à peu au sein de la communauté des *"céïstes"* de part le monde. Car cette communauté existe; quotidiennement, ses membres échangent des idées, transmettent des informations, posent des problèmes, proposent des solutions..., grâce à l'irremplaçable outil que constitue le réseau `internet` qui permet à une station de travail localisée au cœur du Texas de converser avec sa consœur Bretonne ou Occitane.

Par conséquent, en rédigeant ce livre, nous avons eu pour souci principal de mettre en forme cet ensemble de principes, de techniques et de méthodes qui font maintenant partie de la culture d'un programmeur C averti, et sans lesquels il est très difficile de bien programmer en C. Il comporte évidemment un grand nombre d'exemples de programmes, mais aussi de sessions, réalisées directement sous le système (il s'agit ici du système UNIX), et illustrant le résultat de leur exécution. Nous montrons également divers outils indispensables comme les gestionnaires de programmes, débogueurs symboliques, etc.

Nous avons alors dû faire un choix : soit montrer les logiciels classiques contenu en standard dans les distributions officielles des constructeurs, soit montrer leurs homologues *officieux* couramment utilisés par la communauté des programmeurs. Nous avons opté pour la seconde solution en prenant le parti de mettre en avant l'environnement avec lequel nous travaillons quotidiennement, et tout particulièrement les remarquables logiciels du projet *GNU*.

AVANT-PROPOS À LA SECONDE ÉDITION.

Les implémentations ANSI du langage C, encore marginales lors de la première édition de cet ouvrage, sont maintenant en passe de se généraliser. De même le réseau `internet`, qui disposait principalement, à l'époque, d'une audience universitaire, intéresse aujourd'hui de plus en plus le milieu industriel français de l'informatique. Ces tendances, qui confirment les orientations générales de la première édition, nous ont incité à mettre plus encore l'accent sur la norme ANSI d'une part, et sur les logiciels domaine public et du projet *GNU* de l'autre. De ce fait, le chapitre consacré à la bibliothèque standard a été sensiblement remanié et augmenté, dans le sens d'une plus grande conformité à la norme. C'est également le cas de l'ensemble des exemples de programme, organisés en plus de deux cent cinquante fichiers, et désormais indexés en fin de volume. D'autre part, on trouvera en annexe un guide, aussi complet que possible, de l'ensemble des définitions de l'environnement ANSI du langage C.

Mais la modification majeure de cette édition est l'ajout d'un nouveau chapitre, consacré à la modularisation des programmes C. Il nous a, en effet, semblé important de renforcer l'aspect méthodologique de l'ouvrage. Son but est d'amener progressivement le lecteur jusqu'à un modèle, à la fois simple et opérationnel, de programmation par objets, en ayant comme soucis constant, la maintenabilité et l'extensibilité des applications et la réutilisabilité du code source. Cela nous a, à nouveau, conduit à reprendre l'ensemble des exemples de programme, cette fois dans le but de traquer et d'éliminer les fautes d'ordre méthodologique.

REMERCIEMENTS

Je voudrais exprimer ma reconnaissance à tous ceux qui m'ont inspiré, soutenu, ou aidé dans ce travail. Tout d'abord, à Patrick Greussay qui, au début des années 1980, anima le Vax 780 du Gréco Programmation du CNRS sur lequel j'ai découvert le monde UNIX et le langage C. C'est en grande partie grâce à lui, à ses encouragements, et aux nombreux exemples de programme qu'il a mis à la disposition de la communauté informatique française, que j'ai pu devenir peu à peu un *"Céïste Unoxidable"*. Ensuite à Jean Berstel et à Robert Cori, qui m'ont donné l'élan nécessaire pour démarrer cette entreprise. Je remercie également tous les amis et collègues du LaBRI qui m'ont aidé et encouragé, et tout spécialement Pierre Castéran, Viviane Delétage, Jean-Philippe Domenger, Patrick Henry, Michel Pallard, Robert Strandh et Bernard Vauquelin pour les échanges fructueux que nous avons pu avoir et pour les nombreuses remarques et critiques avisées, Olivier Baudon, Jean-Jacques Bourdin, Georges Eyrolles, Frédéric *(Babb's)* Goudal, Pascal Guitton, François Pellegrini et Jean-Guy Penaud pour le soin qu'ils ont apporté à la relecture de cet ouvrage, et Monique Claverie pour ses précieux conseils de rédaction. Je remercie enfin Jean-Michel Coudurier pour ses suggestions judicieuses concernant les effets de bord.

Mais ce travail n'aurait jamais pu aboutir sans la collaboration permanente de Myriam Desainte-Catherine, qui m'a consacré un temps ô combien précieux à travers d'innombrables discussions, conseils, remarques, suggestions, critiques, relectures, corrections..., et a supporté, avec une patience sans limites, l'immixtion dans notre vie privée d'un bien encombrant pensionnaire.

Introduction

Le langage C est le langage naturel du système UNIX; tous deux ont vu le jour vers le début des années 1970, au sein des laboratoires Bell, aux USA.

Les bases du langage C ont été établies par Dennis Ritchie, à partir du langage B, dû à Ken Thompson, l'un des inventeurs d'UNIX. Le langage B, et son successeur le langage C, étaient initialement conçus pour le développement du système UNIX. Pour cette raison, C est muni d'opérateurs et de constructions permettant de travailler au niveau du codage machine des instructions et des données. Il est donc, en ce sens, un langage de *bas niveau.*

Cependant, ses concepteurs ne se sont pas contentés de créer *"encore un autre langage d'assembleur"*. En effet, C appartient à la famille des langages de programmation structurée, héritant comme ses cousins PASCAL et ADA du langage ALGOL. C'est donc un langage typé, disposant de fonctions, de structures de données, de structures de contrôle, etc. En ce sens, il est donc un langage de *haut niveau.*

L'année 1978 voit la naissance officielle du langage C. Ses parents sont Brian Kernighan et Dennis Ritchie, et l'acte de naissance se présente sous la forme d'un livre d'environ 250 pages intitulé *"The C Programming Language"*. Initialement développé sur un PDP-7 dans le cadre du projet UNIX, il connaît un succès quasi-immédiat, et très rapidement, est disponible sur un grand nombre de machines et de systèmes, y compris sur des micro-ordinateurs.

Les clés de ce succès sont multiples :

– *simplicité* : C est un langage de petite taille et il est assez facile d'assimiler rapidement l'ensemble de ses mécanismes;

– *puissance* : C est un langage universel, convenant aussi bien à la programmation système, sa vocation initiale (UNIX en est le meilleur exemple), qu'au codage d'algorithmes complexes, au développement d'interfaces ou au calcul numérique;

– *souplesse* : C est avant tout un langage conçu par des programmeurs pour des programmeurs; la conséquence naturelle est que *tout est possible* en C, la conséquence de cette conséquence étant que C nécessite une grande rigueur et beaucoup de discipline de la part du programmeur;

– *efficacité* : C permet, lorsque c'est nécessaire, d'écrire des programmes adaptés à la structure de la machine hôte (ceci explique une part de son succès sur les micro-ordinateurs);

– *portabilité* : les programmes C sont très souvent directement portables d'une machine à une autre, y compris sur des systèmes différents; la normalisation de la bibliothèque de fonctions, principalement des fonctions d'entrées-sorties, permet d'écrire assez facilement des programmes portables.

Citons également, parmi les originalités du langage, la présence d'un pré-

processeur, l'existence de quarante-cinq opérateurs d'expressions, la possibilité de contrôler l'implémentation des variables, ou de manipuler des adresses de fonctions, etc.

Paradoxalement, sa puissance, sa souplesse et sa concision ont parfois contribué à lui forger, au début des années 80, la réputation d'un langage complexe, peu fiable, difficile à apprendre et difficile à lire. Il est vrai que l'on peut très facilement exhiber des constructions parfaitement illisibles, sortes de *monstres* dont l'existence est inhérente à la puissance d'expression du langage. Mais il s'agit là d'une vision très caricaturale, et il n'est pas raisonnable de juger un langage sur des cas pathologiques.

L'année 1983 est une autre date importante dans l'histoire du langage : C a maintenant 5 ans, il entre dans l'adolescence des langages de programmation. Cela se manifeste par la constitution au sein de l'Institut National Américain de Normalisation, l'ANSI (*American National Standards Institute*), d'un groupe baptisé *X3J11* et chargé de définir et d'adopter une norme du langage C.

Ce travail aboutit fin 1988, et C s'appelle désormais C ANSI, ou C *standard*; il est adulte. Cet événement s'accompagne de la publication de la seconde édition du livre *The C Programming Language*, qui devient le manuel "officiel" du langage C standard.

Les modifications apportées au langage C d'origine, appelé désormais C *traditionnel*, ne sont pas très nombreuses, mais ont considérablement amélioré le langage.

Une des plus importantes est l'ajout d'un mécanisme de contrôle de type entre les paramètres effectifs d'un appel de fonction et les paramètres formels de la fonction appelée. C étant un langage qui favorise la compilation séparée et le découpage d'un programme en de nombreux fichiers, cet apport était indispensable.

Un autre apport également très important est la définition dans la norme de l'ensemble des fonctions constituant la bibliothèque standard du langage, et devant, par conséquent, être intégrées à toutes les implémentations revendiquant le label C ANSI. La majeure partie de ces fonctions était déjà présente dans les implémentations traditionnelles, mais leur mise en œuvre pouvait varier d'un système à l'autre.

Enfin, la norme ANSI a validé diverses fonctionnalités, comme par exemple la définition de constantes symboliques, qui au fil des ans étaient devenues des standards de fait.

Comme nous l'évoquions au début de cette introduction, l'un des atouts majeurs du langage C est son adéquation au système UNIX. Tout d'abord, le fait que ce dernier soit écrit en C facilite considérablement la programmation de logiciels interagissant avec le système. En particulier, la bibliothèque standard sous UNIX comporte un ensemble de fonctions permettant de mettre en œuvre tous les appels systèmes directement depuis un programme C. De plus, dans les années 80, il était relativement facile d'obtenir les sources du système UNIX BSD. Aujourd'hui, il existe au moins deux versions domaine public d'UNIX disponibles avec leurs sources. On peut citer à ce titre le système LINUX, qui est une implémentation remarquable d'UNIX SYSTEM V sur architecture 386. La possibilité d'obtenir les sources de ces distributions constitue un avantage considérable pour l'apprentissage de la programmation système en C.

L'environnement de développement UNIX standard est très fourni : géné-

rateurs de programmes comme LEX et YACC, bibliothèques d'interfaces pour terminaux vidéo, débogueur symboliques, formateurs...

Cet environnement s'est enrichi, vers la fin des années 1980, d'une collection de logiciels mis à la disposition de l'ensemble des programmeurs par les programmeurs eux-mêmes. Ces logiciels, disponibles avec leurs sources sur le réseau international `internet`, et dont la quasi-totalité est écrite en C, ont concrétisé l'émergence d'une *communauté internationale des programmeurs* UNIX, et par voie de conséquence, des programmeurs C.

Citons parmi les exemples les plus remarquables :

- le projet *GNU* incluant le meilleur environnement de programmation en C actuellement disponible – c'est le support que nous avons utilisé pour effectuer le présent travail;
- le gestionnaire de fenêtres *X Window* disponible à l'heure actuelle sur la quasi-totalité des stations de travail;
- les nombreux logiciels d'administration de réseau, de traitement de texte, de traitement d'images, de messagerie...

Quel est l'avenir du langage C aujourd'hui? Durant la vingtaine d'années écoulée depuis sa conception initiale vers 1972, il faut reconnaître qu'il n'a, somme toute, que très peu évolué. Il n'est pas devenu pour autant un langage dépassé. A l'instar de LISP, son aîné d'environ 15 ans, sa puissance d'expression et sa simplicité intrinsèque lui confèrent une grande capacité d'adaptation. L'exemple le plus convaincant concerne l'émergence du paradigme de programmation par objets qui est une des évolutions majeures en matière de programmation de ces dix dernières années : c'est au sein des langages C et LISP qu'ont été développées les *extensions "objet"* les plus couramment utilisées à l'heure actuelle. Par conséquent on peut penser que le langage C a encore de beaux jours devant lui, et que, pourquoi pas, il survivra peut-être même au système UNIX.

Venons en maintenant, à l'organisation de cet ouvrage. Il est découpé en trois parties consacrées respectivement : aux constructions de base du langage, aux méthodes et à l'environnement de programmation, et à l'interface avec le système UNIX.

– Le premier chapitre, est une présentation sommaire de la programmation en langage C; l'accent y est mis sur l'utilisation des expressions et des fonctions, et il sera nécessaire de parcourir une vingtaine de pages pour rencontrer la première affectation de variables.

– Les chapitres 2, 3 et 4 sont consacrés respectivement, aux mécanismes de déclaration, à la construction des expressions, et à celle des instructions; sans pour autant formaliser à outrance, nous nous sommes attachés à présenter rigoureusement ces différentes structures, en évoquant au besoin les constructions algébriques sous-jacentes.

– Le chapitre 5 décrit le fonctionnement du préprocesseur; nous insistons particulièrement sur les possibilités qui s'offrent au programmeur à travers l'utilisation de cet outil, mais aussi sur les nombreux dangers qui le guettent; les nombreux exemples que nous donnons devraient permettre d'exploiter les premières tout en évitant les seconds.

– Le chapitre 6 traite de l'utilisation des pointeurs qui sont les constructions les plus puissantes du langage, mais aussi les plus délicates à manipuler.

– Le chapitre 7 présente les principales fonctions de la bibliothèque standard;

on y trouvera un panorama des problèmes classiques que permettent de traiter ces fonctions.

– Le chapitre 8 aborde les principaux aspects de la modularisation des programmes C, à travers l'abstraction de constantes litérales, la factorisation du code et le masquage de l'implémentation; il présente une méthode d'écriture de modules réutilisables, basée sur la programmation par objets.

– Les chapitres 9 à 13 sont consacrés à l'interface avec le système UNIX et à la mise en œuvre des appels système. Le chapitre 9 présente sommairement quelques principes de base régissant le mécanisme d'appel système. Les suivants traitent respectivement, des mécanismes d'entrée-sortie, de l'interface avec le système de fichiers, de la manipulation des processus, et de la communication entre processus.

Première partie

Structure du langage C

Chapitre 1

Introduction à la programmation en langage C

1.1 Quelques fonctions et c'est tout un programme

La construction fondamentale d'un programme en langage C est la **fonction**. Il existe, dans l'environnement de programmation standard, une centaine de fonctions prédéfinies, directement utilisables dans un programme. Cet ensemble de fonctions constitue la **bibliothèque standard** du langage C. Initialement, l'implémentation des fonctions de la bibliothèque était fortement dépendante du système hôte. Ces fonctions sont maintenant définies de façon précise dans la norme ANSI, et intégrées à toute **implémentation standard**, c'est-à-dire conforme à la norme.

La fonction standard la plus fréquemment utilisée est certainement la fonction `printf`, recopiant sur la sortie standard du programme, en principe l'écran du terminal, une chaîne de caractères reçue en paramètre. Dans un programme C, une **chaîne de caractères** peut être définie au moyen d'une suite quelconque de caractères écrite entre guillemets; les guillemets servent à délimiter le début et la fin de la chaîne. Par exemple, les mots

```
"abcde"    "printf"    "67894321"    "a*.c"
```

sont des chaînes de caractères.

Pour appeler une fonction, il suffit de faire suivre son nom d'une paire de parenthèses, la liste de ses éventuels **paramètres** étant placée entre les deux parenthèses. La phrase

```
printf("printf peut afficher ceci.\n")
```

est un **appel de fonction** demandant à la fonction `printf` de faire afficher son paramètre, ici la chaîne de caractères `printf peut afficher ceci.\n`, sur la sortie standard. On notera que pour écrire la phrase précédente, on a besoin, dans le texte en français, de deux styles de caractères différents. C'est exactement pour la même raison qu'on a besoin, en C, du caractère spécial

guillemet pour délimiter une chaîne de caractères. Le symbole \ est également un caractère spécial, signifiant : ATTENTION, LE CARACTÈRE QUI ME SUIT N'EST PAS DANS SON ÉTAT NORMAL. Par exemple le caractère **n** à la suite d'un \ n'est pas la lettre *n* mais le caractère spécial de fin de ligne, ou *newline*[1] dont on notera la valeur ↵.

Un appel de fonction ne constitue pas à lui tout seul un programme C. Pour pouvoir l'utiliser, il faut tout d'abord le transformer en **instruction**. Cela se fait tout simplement en le faisant suivre d'un point-virgule qui est le caractère de ponctuation terminant chaque instruction. Donc, la phrase

```
printf("printf peut d'afficher ceci.\n");
```

est une instruction du langage C.

Une instruction doit être placée dans une **fonction**. Il est bien sûr possible de placer plusieurs instructions dans une même fonction. Définir une nouvelle fonction consiste en fait à donner un nom, celui de la fonction, à la suite d'instructions qu'elle renferme. Cette construction doit, elle aussi, respecter une syntaxe particulière. Par exemple, pour construire une fonction, de nom **afficher**, constituée de l'instruction précédente et de l'instruction

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

on écrit :

```
afficher()
{
    printf("printf peut afficher ceci.\n");
    printf("            --------\n");
}
```

La définition commence par le nom de la fonction, sur l'exemple **afficher**, suivi d'une paire de parenthèses. Cette partie est appelée l'**en-tête** de la fonction. La liste des instructions est placée à l'intérieur d'une paire d'accolades ({ }). De cette façon, on peut en déterminer sans ambiguïté le début et la fin. Une liste d'instructions placée entre accolades est appelée un **bloc**. La fonction **afficher** constitue presque un programme C. Elle est syntaxiquement correcte; elle est formée de deux instructions, chacune effectuant un appel à la fonction **printf**. Un problème demeure cependant : la fonction **afficher** n'est jamais appelée.

Pour remédier à cela on définit une fonction initiale, dont le nom est **main**, et qui sera appelée automatiquement lors du lancement du programme. Tout programme C doit en posséder une et une seule.

Nous sommes donc maintenant capables d'écrire un programme C complet, que nous allons placer dans le fichier **premier_programme.c** Voici le contenu de ce fichier :

```
════════ premier_programme.c ════════════════════
/*
  Mon premier programme (ou quasiment...)
 */
```

[1] La définition de ce caractère pose toujours un problème du fait qu'il peut être associé à deux caractères différents, les caractères CR ou RETURN et LF ou LINEFEED. Par défaut, sous UNIX, les deux touches RETURN et LINEFEED du clavier sont associées au caractère \n, et l'écriture du caractère \n affiche la suite *CR/LF* à l'écran.

```
#include <stdio.h>

main()
{
    afficher();
}

afficher()
{
    printf("printf peut afficher ceci.\n");
    printf("                  --------\n");
}
```
premier_programme.c

Remarque 1 *Les lignes du programme ci-dessus ne font pas partie du texte source de ce livre. Elles ont été automatiquement insérées par le système de traitement de texte que nous avons utilisé*[2]*, et ce directement depuis le fichier source du programme. De manière générale, toute portion de texte placée entre une* marque de début *de la forme*

$\langle nom.c \rangle$

et une marque de fin *de la forme*

$\langle nom.c \rangle$

visualise le contenu exact du fichier $\langle nom.c \rangle$*. Chacun des fichiers ainsi présentés a été compilé et testé.*

Le programme de l'exemple précédent est composé de deux fonctions; le déroulement de son exécution est visualisé sur la figure 1.1; chaque boîte représente l'entrée dans une fonction. On remarque également en tête du fichier un commentaire le décrivant succinctement. Les deux caractères `/*` (resp. `*/`) signifient *début de commentaire* (resp. *fin de commentaire*).

1.2 Du programme source au fichier exécutable

1.2.1 Compilation et édition de liens

Le fichier `premier_programme.c` est appelé le fichier **source** du programme. L'étape suivante consiste à construire à partir de ce fichier source un fichier exécutable. Les différentes étapes de cette construction enchaînent l'exécution :

- du **compilateur** C qui traduit le programme source en un programme équivalent en *langage machine*;

[2]Il s'agit du système traitement de texte TEX conçu et réalisé par Donald E.Knuth, dans la version LATEX de Leslie Lamport. Ces deux logiciels, ainsi que la multitude de bibliothèques et outils associés (éditeurs de figures, générateur d'index, traducteurs PostScript...), font partie du domaine public.

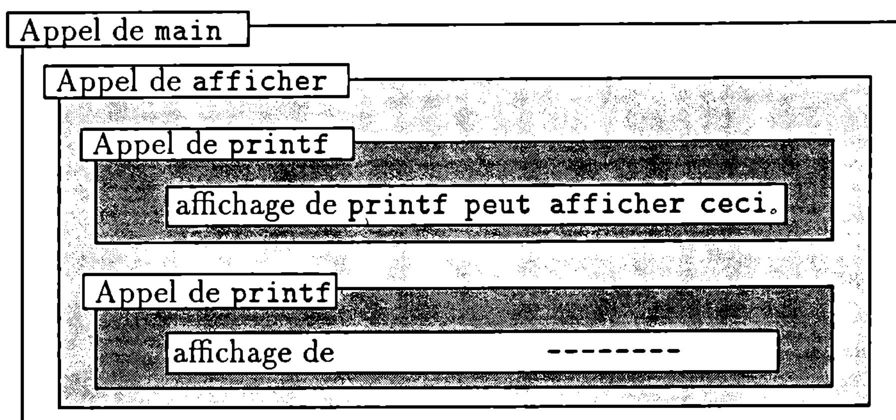


Figure 1.1: Déroulement du programme `premier_programme.c`

– de l'**assembleur** qui construit un fichier appelé fichier **objet** contenant le *code machine* correspondant;
– de l'**éditeur de liens** qui construit le **programme exécutable** à partir du programme objet.

L'invocation du compilateur peut varier suivant les systèmes. Sous le système utilisé ici, à savoir le système UNIX, la commande CC(1)[3] enchaîne automatiquement les différentes étapes intervenant dans la construction de l'exécutable. Ainsi, pour compiler[4] le programme précédent, on tapera simplement :

```
$
$ cc premier_programme.c -o premier_programme
$
```

Les trois lignes ci-dessus sont extraites d'une session de travail sous UNIX. Tous les exemples de session présentés dans cet ouvrage ont été réalisés sous le système et insérés en l'état dans le texte source. Ils sont signalés au moyen d'une barre verticale sur le bord gauche de la page. La première ligne de l'exemple commence par le caractère `$`; c'est le caractère affiché par l'interprète de commandes SH(1)[5] du système UNIX pour inviter à la frappe d'une commande – on parlera de caractère d'invitation ou de *prompt*. Si on tape un *retour chariot*, l'interprète réaffiche le prompt sur la ligne suivante. La commande

`cc premier_programme.c -o premier_programme`

[3]Le manuel du système UNIX est organisé en chapitres, chaque chapitre étant lui-même organisé en pages. Le chapitre 1 décrit les commandes, le chapitre 2 les appels système, le chapitre 3 les fonctions de bibliothèque, etc. La notation CC(1) renvoie à la page du chapitre 1 intitulée CC, et décrivant la commande `cc`. Au sein de chaque chapitre, les pages sont rangées par ordre alphabétique.

[4]On emploie souvent de façon abusive le terme *"compiler"* pour désigner l'ensemble des différentes étapes de compilation, assemblage, et édition de liens...

[5]Il existe sous UNIX plusieurs interprètes de commandes, ou *shells*. Nous avons opté pour l'interprète `sh`, plus simple à utiliser que son homologue l'interprète `csh`, et disponible en standard sur tout système UNIX. De plus, il existe une version publique de cet interprète, à savoir l'interprète `bash` du projet *GNU* intégrant des mécanismes puissants d'édition interactive de commandes.

signifie :

> *"construire avec le fichier* `premier_programme.c` *un exécutable de nom* `premier_programme`*"*.

La compilation s'est déroulée sans erreur; lorsque la commande `cc` nous rend la main, il est possible de vérifier que l'exécutable a été créé au moyen de la commande UNIX **LS(1)** :

```
$ ls -l premier_programme*
-rwxr-xr-x  1 achille     32768 Aug 30 09:27 premier_programme
-rw-r--r--  1 achille       202 Aug 30 08:23 premier_programme.c
$
```

Lorsqu'il lit la ligne de commande `ls -l premier_programme*`, le shell construit la liste de tous les fichiers du répertoire courant dont le nom commence par `premier_programme`, puis lance la commande `ls` avec cette liste en argument. Celle-ci, invoquée avec l'option `-l`, fournit un compte-rendu de chacun des fichiers de la liste : droits d'accès, propriétaire, taille, date de création... La plupart de ces concepts seront développés dans la suite de cet ouvrage.

Le fichier `premier_programme` que nous venons de créer est une commande à part entière. Pour la faire exécuter, il suffit de taper son nom :

```
$ premier_programme
printf peut afficher ceci.
               --------
$
```

Les exemples présentés dans cet ouvrage ont été compilés avec le compilateur `gcc`, qui est un compilateur développé au sein du projet *GNU*, conduit par Richard Stallman. Il s'agit d'un compilateur distribué gratuitement avec ses sources et sa documentation. Sur de nombreux aspects : qualité du code généré, rapidité, traitement des erreurs, optimisation du code... il est de meilleure qualité que la majeure partie des compilateurs C du marché. Il a été porté sur de nombreuses machines : Sun, SGI Iris, Encore, RS6000, HP320, Convex, Vax, Apollo, diverses architectures Intel 386 et Motorola 68k... et sur plusieurs systèmes. Il supporte la norme ANSI, et diverses extensions que nous avons évité d'utiliser ici[6]. Tout ce que nous dirons concernant la commande `gcc` s'applique également, sauf mention du contraire, à la commande `cc` classique.

La commande `gcc` peut être invoquée avec différentes options. L'une des plus fréquemment utilisées est l'option `-c` demandant que seuls le compilateur et l'assembleur soient appelés. Dans ce cas, le résultat produit est le fichier objet; il reçoit automatiquement le suffixe `.o` :

```
$ gcc -c premier_programme.c
```

[6] La distribution de `gcc` et des autres logiciels du projet *GNU* peut être obtenue en se connectant à travers le réseau `internet`, par `ftp` sous le nom `anonymous` (voir **FTP(1)** et **FTPD(8)**), sur la machine `prep.ai.mit.edu` ou sur un serveur `ftp` en France. Il est possible d'obtenir des informations à l'AFUU : *Association Française des Utilisateurs d'Unix, 11 rue Carnot, 94270, Le Kremlin-Bicêtre*, ou à *Free Software Foundation, 1000 Mass Ave, Cambridge MA 02138, USA*.

```
$ ls -l premier_programme.o
-rw-r--r--  1 achille      268 Aug 30 09:43 premier_programme.o
$
```

Il s'avère très vite indispensable de découper un programme en plusieurs fichiers. Nous allons voir comment on procède à partir du programme précédent, en le découpant en deux fichiers source, l'un contenant la fonction `main` et l'autre la fonction `afficher`.

```
main.c
main()
{
    afficher();
}
main.c
```

```
afficher.c
afficher()
{
    printf("printf peut afficher ceci.\n");
    printf("            --------\n");
}
afficher.c
```

L'exemple suivant montre comment construire un exécutable à partir de ces deux fichiers.

```
$ gcc main.c afficher.c -o afficher
$
$ afficher
printf peut afficher ceci.
            --------
$
```

Lorsqu'un programme est découpé en plusieurs fichiers source, il est utile de conserver les fichiers objet correspondants. En effet, lors de la mise au point du programme, il est possible de recompiler seulement les fichiers source modifiés. Pour cela, on utilise l'option `-c` évoquée plus haut :

```
$ gcc -c main.c
$ gcc -c afficher.c
$ ls *.o
afficher.o     main.o
$
```

Supposons maintenant qu'on désire changer le message écrit par la fonction `afficher`; seul le fichier `afficher.c` nécessite d'être modifié :

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

```
afficher()
{
    printf("Savez vous que ");
    printf("printf est capable d'afficher ceci ?\n");
}
```

afficher.c

La reconstruction de l'exécutable s'effectue à partir du nouveau fichier source `afficher.c` et de l'ancien fichier objet `main.o` :

```
$ gcc -c afficher.c
$ gcc main.o afficher.o -o afficher
$
$ afficher
Savez vous que printf est capable d'afficher ceci ?
$
```

Il ne faut pas hésiter à découper un programme en de nombreux fichiers, même de petites tailles. Chaque composante logique du programme doit être définie dans un fichier spécifique. Nous verrons dans le chapitre 8 comment organiser ces fichiers pour définir des modules indépendants et réutilisables. Cette discipline oblige le programmeur à structurer l'application qu'il développe. De plus, le système UNIX encourage cette façon de faire en fournissant divers outils de gestion de programmes, comme l'indispensable commande MAKE(1) dont nous allons présenter rapidement le fonctionnement en reprenant l'exemple précédent.

1.2.2 Commande `make` et fichier `Makefile`

La commande `make` recherche dans le répertoire de travail courant un fichier de nom `Makefile` contenant la description des dépendances entre les différents fichiers intervenant dans la construction du programme. Le fichier `Makefile` du programme `afficher` doit contenir les informations suivantes :

- `afficher` est construit avec `main.o` et `afficher.o`;
- `afficher.o` est construit avec `afficher.c`;
- `main.o` est construit avec `main.c`.

Voici une façon d'écrire le fichier `Makefile` correspondant.

Makefile

```
CC = gcc
CFLAGS = -g
OBJ = main.o afficher.o

afficher : $(OBJ)
	$(CC) $(CFLAGS) $(OBJ) -o afficher

afficher.o : afficher.c
main.o     : main.c

clean:
	rm -f $(OBJ) afficher
```

Makefile

Les trois premières lignes sont des définitions de variables. Les deux premières font partie d'un ensemble de variables prédéfinies paramétrant le comportement de la commande make (voir MAKE(1)). La variable CC contient le nom de la commande de compilation qu'on désire utiliser (par défaut, il s'agit de la commande cc). La variable CFLAGS permet de spécifier les options à transmettre par défaut à la commande de compilation. L'option -g est l'option de débogage symbolique (voir 7.4). La troisième ligne définit une nouvelle variable, la variable OBJ, initialisée avec la liste des fichiers objet du programme. Il est également possible de définir des variables dans l'environnement exportable de l'interprète de commandes (par exemple, export CC=gcc sous un interprète de type shell).

La construction $(⟨*var*⟩) permet d'obtenir la valeur de la variable ⟨*var*⟩. Par exemple, compte tenu des définitions précédentes,

```
afficher: $(OBJ)
```

est équivalent à

```
afficher: main.o afficher.o
```

et

```
$(CC) $(CFLAGS) $(OBJ) -o afficher
```

à

```
gcc -g main.o afficher.o -o afficher
```

Ces deux lignes sont respectivement, une spécification de dépendance, et une action associée à cette spécification. Il est en général nécessaire de faire commencer chaque ligne décrivant une action par un caractère de tabulation en première colonne. Plusieurs actions peuvent être associées à une seule spécification.

Lorsque les suffixes des fichiers le permettent, make génère automatiquement l'action associée à une spécification; par exemple il n'est pas besoin d'indiquer comment construire un fichier .o avec un fichier .c. Par défaut, lors de son lancement, make essaie de réaliser la première spécification en parcourant le graphe des dépendances induit par le fichier Makefile. Il suffit par conséquent de taper

```
make
```

et la commande fait le reste :

```
$ rm *.o
$ make
gcc -g   -c main.c -o main.o
gcc -g   -c afficher.c -o afficher.o
gcc main.o afficher.o -o afficher
$
```

L'exécution de rm *.o détruit tous les fichiers du répertoire courant dont le nom se termine par .o (voir SH(1) et RM(1)). L'exécution de make enchaîne les différentes étapes de compilation et d'édition de liens nécessaires à la construction de afficher.

Lorsqu'on ne modifie que certains fichiers de l'application, la commande make génère le minimum d'actions nécessaire pour reconstruire l'exécutable. Elle utilise pour cela la date de la dernière modification des fichiers. Cette information, propre à chaque fichier, est mise à jour automatiquement par le système lors d'une modification. Il est possible de réactualiser cette date au moyen de la commande **TOUCH(1)**. Voici différents exemples d'exécution de make :

```
$ rm afficher
$ make
gcc afficher.o main.o -o afficher
$
$ touch afficher.c
$ make
gcc -g   -c afficher.c -o afficher.o
gcc afficher.o main.o -o afficher
$
$ make
'afficher' is up to date.
$
```

On notera enfin l'option -MM du compilateur générant la liste des dépendances d'un ensemble de fichiers :

```
$ gcc -MM *.c
afficher.o : afficher.c
main.o : main.c
$
```

Cette option est très utile, soit pour générer un squelette de fichier Makefile, soit pour vérifier la validité d'un fichier Makefile déjà défini. Avec le compilateur C traditionnel, on utilisera l'option -M, l'option -MM n'étant pas définie[7].

En conclusion, nous insisterons sur le fait que l'utilisation de la commande make n'est pas réservée aux spécialistes chevronnés, développant d'énormes applications. Il est très important, dès l'écriture de ses premiers programmes, d'acquérir le réflexe suivant :

1) pour chaque programme : création d'un répertoire contenant ses fichiers source et un fichier Makefile;
2) pour chaque nouveau fichier source ajouté dans le répertoire : mise à jour du fichier Makefile en conséquence.

Remarque 2 *La commande* gcc *accepte en arguments des fichiers de natures différentes : fichiers source C, programmes objet, mais aussi programmes en assembleur, en fortran, bibliothèques de fonctions... Elle identifie chaque fichier au moyen de son suffixe :* .c, .o, .s, .f, .a, *etc. Pour cette raison, ce suffixe est indispensable et ne peut être modifié.*

[7] L'option -MM ne prend en compte que les fichiers de l'utilisateur, alors que l'option -M parcourt également les fichiers en-tête du système, générant ainsi des dépendances inutiles.

1.3 Type et valeur des expressions

Nous allons maintenant revenir au langage C. Pour l'instant, nous avons utilisé seulement deux objets, la fonction et la chaîne de caractères. Ils font partie des éléments de base du langage, comme les nombres ou les caractères. Ces éléments sont utilisés pour former des **expressions** comme par exemple :

`256*1024`	:	multiplication de 256 par 1024;
`0.7/1.3`	:	division de 0.7 par 1.3;
`1.0 - cos(1.57)`	:	soustraction de 1.0 et de ***cos*** 1.57;
`'1' < 'A'`	:	comparaison des caractères `1` et `A`.

Ces expressions, comme toute expression C correcte, possèdent un **type** et une **valeur**. Le type et la valeur des quatre expressions précédentes sont respectivement :

entier	262144
réel flottant	0.538462
réel flottant	0.999204
$\{0,1\}$	1

La valeur 1 de la dernière expression code la valeur booléenne *vrai*; la valeur *faux* est codée par 0.

Les expressions sont semblables à un jeu de construction. On dispose d'éléments de base, ici des constantes entières, des appels de fonction..., et à l'aide des mécanismes d'assemblage que constituent les **opérateurs** comme `*` ou `<`, on bâtit de nouvelles expressions plus complexes. On peut alors réutiliser ces nouvelles expressions comme éléments de base d'une autre construction.

Les opérateurs permettant de construire des expressions sont très nombreux en C. Voici la liste des plus classiques :

- `+ - * /` : opérations arithmétiques usuelles;
- `%` : reste de la division euclidienne;
- `> >= < <= == !=` : comparaisons, resp. $> \geq < \leq = \neq$;
- `&& ||` : connecteurs logiques *et* et *ou*.

Un autre opérateur, très original, est l'opérateur conditionnel

`?:`

qui permet, avec trois expressions $\langle e\rangle$, $\langle e_1\rangle$ et $\langle e_2\rangle$, de construire l'expression

$$\langle e\rangle \texttt{ ? } \langle e_1\rangle \texttt{ : } \langle e_2\rangle$$

Cette expression vaut $\langle e_1\rangle$ si $\langle e\rangle$ est différent de zéro et $\langle e_2\rangle$ sinon. Par exemple l'expression

```
(rand() % 2 == 0) ? 1 : -1
```

vaut 1 si la valeur retournée par la fonction `rand` est paire et -1 sinon (la fonction `rand` a pour valeur un nombre entier pseudo-aléatoire compris entre 0 et, au minimum, $2^{15}-1$). Les expressions $\langle e_1\rangle$ et $\langle e_2\rangle$ ne sont pas obligatoirement des expressions de type numérique; par exemple, l'expression

```
(rand() % 2 == 0) ? "pair" : "impair"
```

est de type *chaîne de caractères*.

1.4 Déclaration des fonctions

1.4.1 Type d'une fonction

Un appel de fonction étant une expression, il possède un type et une valeur. Par conséquent une fonction doit également posséder un type et être capable de retourner, lors de son exécution, une valeur de ce type. Voici un nouvel exemple de définition de fonction, comportant la spécification de son type et de sa valeur de retour.

```
double pi()
{
    return 3.1415926535897931;
}
```

Le type de la fonction se place devant son nom : la fonction `pi` est de type `double`, c'est-à-dire réel double précision. La valeur de retour se place à la suite du mot `return` : la fonction pi retourne la valeur 3.1415926535897931. La phrase

```
return 3.1415926535897931;
```

est une instruction; par conséquent elle est ponctuée par un point virgule.

La fonction `pi` peut être réutilisée dans la construction d'une autre expression; par exemple, `0.5*pi()` est une expression de type `double` et de valeur 1.5707963267948966. Le mot `return` peut être suivi d'une expression quelconque; la seule contrainte à respecter est très naturelle : le type de l'expression retournée et celui de la fonction doivent être les mêmes :

```
double pisurdeux()
{
    return 0.5*pi();
}
```

Les précédentes fonctions que nous avons construites n'avaient pas de type explicitement déclaré. De telles fonctions sont implicitement de type `int`, c'est-à-dire nombre entier. La fonction `afficher` présentée page 12 aurait donc pu être écrite de façon équivalente :

```
int afficher()
{
    ...
}
```

Cependant, il n'est pas logique qu'une fonction qui ne retourne aucune valeur soit de type entier. On la déclarera de type `void`, c'est-à-dire *vide* :

```
void afficher()
{
    ...
}
```

La possibilité d'une part de définir des fonctions de type quelconque, et d'autre part de découper un programme en plusieurs fichiers source, pose un nouveau problème : si une fonction f est définie dans un fichier F_1 et utilisée dans un fichier F_2, il est nécessaire d'indiquer dans F_2 le type de f. Cela se fait au moyen d'une déclaration référençant la fonction. La syntaxe de cette

déclaration est calquée sur celle de la définition de fonction. Elle se compose du type de la fonction à laquelle on fait référence, suivi de son nom et d'une paire de parenthèses. Tout comme les instructions, les déclarations sont ponctuées par un point virgule.

La déclaration peut être placée de différentes façons :

- au début du fichier, elle est visible dans tout le fichier;
- au début d'un bloc, elle est visible dans tout le bloc (y compris dans les éventuels sous-blocs de ce bloc).

La fonction `main` du programme `afficher` peut se récrire en :

```
extern void afficher();

main()
{
    afficher();
}
```

Le mot-clé `extern` indique que cette déclaration est une **référence** à une définition effectuée dans un autre fichier. Il est également nécessaire de référencer une fonction dans le fichier où elle est définie lorsque sa définition est située après son utilisation. Ainsi, dans le fichier suivant, l'utilisation de la fonction `pi` précède sa définition :

```
======== erreur_decl.c ========================================
#include <stdio.h>

main()
{
    printf("Valeur approchee de pi/2: %f\n", pi()/2.0);
}

double pi()
{
    return 3.1415926535897931;
}
======================================== erreur_decl.c ========
```

Sa compilation provoque un message d'erreur :

```
$ gcc -g -c erreur_decl.c
erreur_decl.c:9: warning:type mismatch with previous external decl
erreur_decl.c:5: warning: previous external decl of 'pi'
erreur_decl.c:9: warning: type mismatch with previous implicit
                          declaration
erreur_decl.c:5: warning: previous implicit declaration of 'pi'
erreur_decl.c:9: warning: 'pi' was previously implicitly declared
                          to return 'int'
$
$ cc -c erreur_decl.c
"erreur_decl.c", line 9: redeclaration of pi
$
```

En effet le compilateur C effectue une déclaration implicite de type entier chaque fois qu'une fonction est utilisée sans avoir été préalablement déclarée. Lorsque la fonction `pi` est utilisée pour la première fois à la ligne 5, le compilateur la déclare implicitement de type entier. La déclaration explicite de la fonction `pi` à la ligne 8 de type `double` constitue par conséquent une redéfinition de type. Une erreur du même ordre se trouve explicitée dans le fichier source `autre-erreur-decl.c`. Cette fois, la fonction est référencée avant son utilisation, mais les deux définitions sont incompatibles.

```
autre_erreur_decl.c
int pi();

double pi()
{
    return 3.1415926535897931;
}
autre_erreur_decl.c
```

Le compilateur gcc signale maintenant une erreur fatale, au lieu d'un simple message d'avertissement comme dans le cas précédent.

```
$ gcc -g -c autre_erreur_decl.c
autre_erreur_decl.c: In function pi:
autre_erreur_decl.c:4: conflicting types for 'pi'
autre_erreur_decl.c:1: previous declaration of 'pi'
$
$ cc autre_erreur_decl.c -c
"autre_erreur_decl.c", line 4: redeclaration of pi
$
```

Cette tolérance du compilateur gcc envers certaines déclarations inconsistantes ne doit pas être exploitée. En effet, il y a de fortes chances que le programme généré soit sémantiquement incorrect, ce qui est le cas de `erreur_decl` :

```
$ erreur_decl
Valeur approchee de pi/2: 0.000000
$
```

L'erreur est ici due au fait que le codage des entiers et celui des réels diffèrent en machine.

Afin de minimiser les risques d'erreurs, il est préférable de déclarer toutes les fonctions avant leur utilisation, même lorsque la syntaxe C ne l'impose pas.

1.4.2 Prototype d'une fonction

Les fonctions que nous avons définies jusqu'à présent ne comportent pas de paramètre. Par contre, certaines d'entre elles font appel à des fonctions de bibliothèque (`printf`, `sin`) appelées avec des paramètres. Nous allons voir maintenant comment définir une fonction avec paramètres, en considérant l'exemple de la fonction `fsomme` retournant la somme de deux réels flottants, c'est-à-dire

du type `float`, passés en paramètres. Il existe deux syntaxes possibles qui sont les suivantes :

```
float fsomme(a, b)
    float a;
    float b;
{
    return a + b;
}
```

et

```
float fsomme(float a, float b)
{
    return a + b;
}
```

Avec la première forme, la liste des **paramètres formels** a et b est placée à l'intérieur des parenthèses et la liste des déclarations correspondantes entre les parenthèses et le début du bloc. Cette syntaxe est celle du langage C traditionnel. Elle tend à disparaître au profit de la seconde forme, celle de la norme ANSI, à la fois plus concise et plus puissante. C'est également la forme utilisée dans le langage C++ qui est le langage de programmation par objets défini au-dessus de C. Les compilateurs C standard acceptent encore l'ancienne forme pour des raisons de compatibilité.

Remarque 3 *Les exemples présentés dans cet ouvrage sont conformes à la syntaxe du langage* C *norme* ANSI, *encore appelé* C **standard**, *dont nous allons voir l'importance ci-dessous. Si le compilateur hôte d'un système n'est pas* ANSI, *nous recommandons vivement d'installer* gcc. *Sur un micro-ordinateur IBM-PC ou compatible, nous recommandons d'installer* `linux` *qui est un système* UNIX *domaine public, de très bonne qualité, et supportant les logiciels du projet GNU.*

Remarque 4 *Il existe, dans le domaine public, divers outils permettant de convertir des programmes écrits selon la syntaxe traditionnelle dans la syntaxe* ANSI *et inversement.*

Une fonction doit être appelée avec exactement le même nombre de **paramètres effectifs** qu'il y a de paramètres formels dans sa définition. De plus, chaque paramètre effectif doit être une expression de même type que le paramètre formel correspondant, ou d'un type compatible (`float` et `double` par exemple). Lors de l'appel de la fonction, le paramètre formel reçoit la valeur du paramètre effectif correspondant, qui peut être une expression quelconque. Par exemple, les expressions

```
fsomme(3.0, 2.0)
fsomme(1.0, 1.0/fsomme(1.0, pi()))
```

valent respectivement 5.0 et 1.241453 (approximation par un flottant du nombre réel $1 + \frac{1}{1+\pi}$). Voici une autre fonction calculant le maximum de deux nombres réels :

```
float fmax(float a, float b)
{
    return (a > b) ? a : b;
}
```

Par exemple, l'expression `fmax(pi()*pi(), 10.0)` est de type réel flottant et vaut 10.

Pour utiliser la fonction `fmax` dans un autre fichier, il suffit de la déclarer de la façon suivante :

```
extern float fmax(float, float);
```

Entre les parenthèses, on retrouve la liste des types des paramètres formels de la fonction. La partie de cette déclaration qui suit le mot clé `extern` est appelée le *prototype* de la fonction `fmax`.

Plus généralement, on appelle **prototype** d'une fonction la déclaration du type des paramètres de cette fonction et de la valeur qu'elle retourne. D'un point de vue pratique, le prototype d'une fonction est constitué de l'en-tête de cette fonction (tout ce qui est situé avant la première accolade ouvrante) dans laquelle sont effacés les noms des paramètres formels. Le prototype de la fonction sans paramètre `afficher` est `void afficher(void)`, et la référence à cette fonction s'écrit :

```
extern void afficher(void);
```

On emploiera parfois abusivement le terme de prototype pour désigner la construction ci-dessus, c'est-à-dire une déclaration, référençant une fonction, et construite à partir d'un prototype.

En C traditionnel, pour utiliser la fonction `fmax`, on déclare simplement

```
float fmax();
```

C'est certainement la lacune la plus grave du langage C traditionnel : il n'y a pas de contrôle de compatibilité de type entre les paramètres effectifs et les paramètres formels correspondants. Par conséquent, l'expression

```
fmax(pi()*pi(), 10)
```

dans laquelle le second paramètre est une constante entière, ne génère aucune erreur de syntaxe à la compilation mais produira un résultat incorrect lors de son évaluation, du fait que les codages en machine de l'entier `10` et du réel `10.0` sont différents. C'est certainement la raison principale pour laquelle il faut adopter la norme ANSI. On appliquera d'autre part la règle suivante :

Afin de permettre au compilateur d'effectuer le contrôle des paramètres effectifs des appels de fonction, on placera en tête de chaque fichier le prototype de toutes les fonctions externes utilisées (cette règle sera reformulée avec l'utilisation des fichiers en-tête – voir page 161).

1.5 La fonction printf

La fonction `printf` avec laquelle nous avons commencé ce chapitre est un cas très particulier. En effet, elle peut être appelée avec un nombre quelconque, non

nul, de paramètres; de plus, excepté le premier qui est toujours une chaîne de caractères, ces paramètres peuvent être de types variés : entiers, réels, chaînes de caractères... Par exemple on peut effectuer l'appel suivant :

```
printf("Le nombre %f est une approximation de pi.\n", pi())
```

Le premier paramètre est la chaîne de caractères obligatoire et le second est le nombre réel retourné par la fonction `pi`. Contrairement aux cas précédents, la chaîne n'est pas affichée telle quelle par `printf`; elle est auparavant interprétée, et certaines suites de caractères sont récrites différemment. Ces suites de caractères sont des **spécifications de format**; elles commencent toutes par le caractère `%`. Dans l'exemple précédent, on trouve la suite de caractères `%f`, spécification de format signifiant :

> *"Le paramètre effectif correspondant est un nombre réel flottant dont la valeur doit être écrite à la place des deux lettres `%f`."*

Cet appel de fonction produira le message :

```
Le nombre 3.141593 est une approximation de pi.
```

Il existe diverses variantes à cette spécification, comme par exemple `%6.4f`, signifiant :

> *"Le paramètre effectif correspondant est un nombre réel flottant dont la valeur doit être imprimée sur six caractères au minimum, avec au plus quatre chiffres après le point décimal."*

L'exécution de :

```
printf("Le nombre %6.4f est une approximation de pi.\n", pi())
```

produit l'affichage de la chaîne :

```
Le nombre 3.1416 est une approximation de pi.
```

Parmi les autres spécifications de format, citons :

- `%d`: paramètre entier à écrire en base 10;
- `%x`: paramètre entier à écrire en base 16;
- `%c`: paramètre de type caractère;
- `%s`: paramètre chaîne de caractères.

La suite de caractères `%%` est une spécification constante codant le caractère `%` lui-même[8].

Nous allons détailler un exemple un peu plus complet : supposons qu'on veuille faire afficher la valeur d'une variable flottante `rapport` et son approximation par un pourcentage. Par exemple, l'affichage correspondant à une valeur de la variable `rapport` égale à 0.45554 devra avoir la forme suivante :

```
Valeur: 0.455540
Taux  : 46%
```

Ce résultat est obtenu en appelant `printf` avec le format

```
"Valeur: %f\nTaux  : %2.0f%%\n"
```

[8] On remarquera qu'il n'est pas possible d'écrire `\%` pour empêcher l'interprétation de `%`; en effet, le caractère d'échappement `\` s'adresse au compilateur alors que l'interprétation de `%` est effectuée à l'exécution par la fonction `printf`.

et les deux expressions rapport et rapport*100.0. Chaque paramètre est évalué avant d'être passé à la fonction printf. Les trois valeurs résultant de cette évaluation sont d'une part la chaîne de caractères

```
Valeur: %f↵Taux  : %2.0f%%↵
```

et d'autre part les réels flottants 0.45554 et 45.554. Le premier paramètre, le format, est découpé en une suite de spécifications et de caractères ordinaires :

Valeur:	‖%f‖	↵Taux :	‖%2.0f‖	‖%%‖	↵

Les spécifications de format sont encadrées par un trait double.

Chaque spécification est remplacée par l'écriture du paramètre correspondant, excepté dans le cas particulier de %% qui n'attend aucun paramètre; les caractères ordinaires sont recopiés tels quels.

La déclaration du type de la fonction printf pose un problème du fait qu'elle accepte un nombre variable de paramètres. On utilise dans ce cas la notation ... encore appelée **ellipse**. Le prototype de printf est :

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

(la fonction printf est de type int car elle retourne le nombre de caractères écrits).

Pour ne pas avoir à recopier ce prototype dans chaque programme utilisant printf, il a été prédéfini dans un fichier de l'environnement noté <stdio.h>[9]. Ce fichier est un fichier **en-tête**; il fait partie des **fichiers en-tête** standard de l'environnement de programmation du langage C. Un fichier en-tête peut être inclus automatiquement dans un autre au moyen d'une **directive** d'inclusion. Une directive n'est pas une instruction du langage mais une commande destinée au **préprocesseur**, qui est un traducteur appelé automatiquement avant le compilateur. Une directive commence par un dièse (caractère #) en première colonne. L'inclusion du fichier en-tête <stdio.h> est commandée par la directive

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

Dorénavant, on placera cette ligne au début de chaque fichier effectuant des entrées-sorties au moyen des fonctions de la bibliothèque standard.

1.6 La fonction main

Un autre cas assez particulier est celui de la fonction main. Nous l'avons jusqu'à présent définie et utilisée comme une fonction sans paramètre. En réalité, la fonction main est appelée avec plusieurs paramètres, qu'il est possible d'ignorer.

Nous avons vu que lorsqu'on tape sous l'interprète de commandes le nom d'un exécutable, celui-ci est chargé en mémoire, créant un processus, dont l'exécution commence par un appel à la fonction main. Si on tape une commande dont le nom est suivi d'**arguments**, ces arguments deviennent les paramètres de main. Du fait qu'une commande peut être invoquée avec un nombre variable d'arguments, ceux-ci ne sont pas transmis à la fonction main les uns à la suite des autres, mais sous la forme d'une liste.

---

[9] Le nom stdio est une contraction de *standard input output*.

Considérons l'exemple suivant : on tape sous l'interprète de commandes la commande :

`ls -l main.c`

Elle est décomposée trois arguments qui sont les mots `ls`, `-l` et `main.c`, le premier argument étant, par convention, le nom de la commande. Il est de tradition d'appeler `argc` et `argv` respectivement le nombre et la liste de ces arguments. L'interprète code la liste des arguments au moyen de la structure de données représentée sur la figure 1.2.

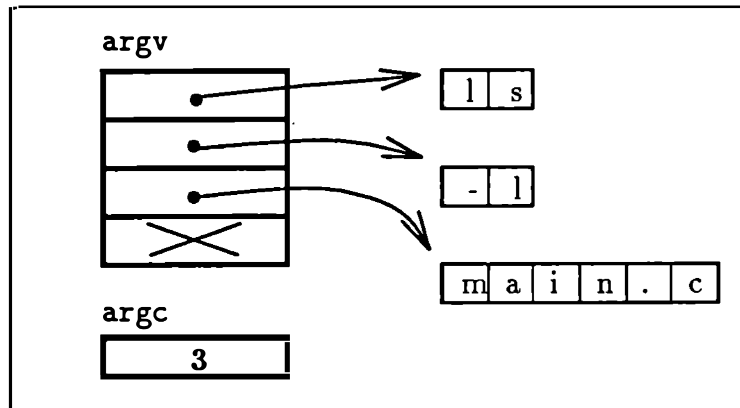


Figure 1.2: Arguments d'une commande

Pour coder la structure de données `argv`, nous allons utiliser une nouvelle construction : le **vecteur**. Un vecteur est une suite de valeurs de même type. Si `suite` est un vecteur, la valeur du premier élément s'écrit `suite[0]`, et celle du $i^{\text{ème}}$ s'écrit `suite[`i`-1]`.

Les arguments d'une commande étant des mots, c'est-à-dire des chaînes de caractères, la liste de ces arguments est un vecteur de chaînes de caractères. La déclaration du vecteur de chaînes de caractères `argv` s'écrit :

```
char *argv[];
```

où `char *` signifie *chaîne de caractères* et `argv[]` précise que `argv` est un vecteur.

La définition standard de la fonction `main` s'écrit par conséquent :

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

Le premier argument reçu étant le nom de la commande, la valeur de `argc` est toujours supérieure ou égale à un.

La commande `Combien` de l'exemple suivant imprime le nombre d'arguments avec lesquels elle est lancée.

```
═════ Combien.c ═════
#include <stdio.h>

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

```
    printf("%d mot%s.\n", argc, (argc > 1) ? "s" : "");
}
```
Combien.c

On retrouve sur cet exemple l'opérateur d'expression conditionnelle, utilisé pour marquer l'accord en nombre de mot.

```
$ Combien y a t il de mots dans cette phrase
10 mots.
$ Combien
1 mot.
$
```

Il est également très facile d'écrire un programme utilisant son propre nom :

Hal.c

```
#include <stdio.h>

main(int argc, char *argv[])
{
    printf("Mon nom est %s.\n", argv[0]);
}
```
Hal.c

Si on renomme programme exécutable au moyen de la commande MV(1), le message affiché est automatiquement modifié. Cette opération ne nécessite pas de recompilation. Sur l'exemple suivant, l'exécution de `mv Hal Personne` change le nom du fichier exécutable `Hal` en `Personne`.

```
$ Hal
Mon nom est Hal.
$
$ mv Hal Personne
$ Personne
Mon nom est Personne.
$
```

L'utilisation du nom d'une commande est très utile pour paramétrer certains messages émis par un logiciel, et notamment les messages d'erreur, avec le nom du logiciel. Voici un exemple illustrant ce comportement : la commande gcc est recopiée sous un nouveau nom dans le répertoire courant, et le message d'erreur émis par cette nouvelle commande reste cohérent :

```
$ gcc
gcc: No input files specified.
$
$ cp /usr/local/bin/gcc ./gnu-cc
$ gnu-cc
gnu-cc: No input files specified.
$
$ rm gnu-cc
$
```

1.7 Variables et effets de bord

1.7.1 Déclaration et initialisation de variables

Lors d'un appel de fonction, les paramètres formels reçoivent les valeurs respectives des paramètres effectifs. Ces paramètres formels sont des **variables**, c'est-à-dire des zones mémoire identifiées par un symbole appelé **identificateur**, et dans lesquelles il est possible de ranger des valeurs. La valeur reçue par le paramètre est une valeur initiale qui peut ensuite être modifiée. La modification de la valeur d'une variable est également appelée un *effet de bord*. Il est naturellement possible de déclarer, dans une fonction, d'autres variables que ses paramètres formels éventuels. La syntaxe de telles déclarations est identique à celle des paramètres formels; les déclarations sont placées au début du bloc dans lequel sont utilisées les variables. Il est de plus possible de préciser, après le nom de la variable, une valeur initiale. Cette valeur peut être une expression quelconque qui sera évaluée lors de l'entrée dans le bloc. La syntaxe d'une déclaration avec initialisation est la suivante :

$\langle type \rangle \quad \langle identificateur \rangle$ = $\langle expression \rangle$;

Considérons le programme `delta.c` suivant :

```
═══════════ delta.c ═══════════════════════════════════
#include <stdio.h>
#include <stdlib.h>

main(int argc, char *argv[])
{
    double a = atof(argv[1]);
    double b = atof(argv[2]);
    double c = atof(argv[3]);

    printf("Delta = %g\n", b*b - 4*a*c);
}
═══════════════════════════════════════ delta.c ═══════
```

Ce programme utilise trois variables double précision, de noms respectivement `a`, `b` et `c`, initialisées avec trois nombres passés en arguments. La spécification de format utilisée, `%g`, affiche un nombre réel en choisissant le format le plus approprié.

```
$ delta 2 3 5
Delta = -31
$
$ delta 2 5 3
Delta = 1
$
```

Chaque argument étant transmis sous la forme d'une chaîne de caractères, il est nécessaire de le convertir en une valeur numérique de type réel double précision

pour effectuer l'initialisation. Cette conversion est réalisée par la fonction de bibliothèque `atof` (abréviation de *Ascii to Float*) dont le prototype est :

```
double atof(char *)
```

Ce prototype est prédéfini dans le fichier en-tête `<stdlib.h>`.

Il est bien sûr possible de déclarer une variable sans initialisation et d'effectuer ensuite une affectation sur cette variable. Cependant, lorsque c'est possible, il est préférable de combiner l'initialisation avec la déclaration.

1.7.2 Affectation de variables

L'**affectation** d'une variable est effectuée au moyen de l'expression d'affectation. L'opérateur de construction de cette expression est le caractère `=`. Par exemple, l'expression

```
Total = Total + 1
```

affecte la valeur de l'expression `Total+1` à la variable `Total`. Nous reviendrons plus longuement dans la suite de cet ouvrage sur ces expressions très particulières du langage C que sont les expressions à effet de bord.

1.7.3 Paramètres passés par référence

Le passage par référence permet d'effectuer des effets de bord sur les paramètres effectifs d'une fonction. Nous avons vu que, par défaut, seulement la valeur du paramètre est transmise à la fonction appelée. Il est également possible de passer la référence du paramètre, en l'occurrence son adresse en mémoire. Pour cela, on utilise l'opérateur `&` qui se place devant la variable à référencer.

Nous ne verrons pas dans cette introduction comment manipuler la référence au niveau du paramètre formel de la fonction. Nous allons simplement donner un exemple d'appel de fonction contenant des paramètres effectifs passés par référence, à travers l'utilisation de la fonction `scanf` de la bibliothèque standard.

La fonction `scanf` est la fonction symétrique de `printf` : elle effectue des lectures sur l'entrée standard. Son premier paramètre est une chaîne de format semblable à celle traitée par `printf`; viennent ensuite les références des variables où seront rangées les valeurs à lire. Cette fonction est assez délicate à utiliser et sa mise en œuvre souvent périlleuse. Aussi nous bornerons-nous à des appels du type :

`scanf("%d", &n)`	:	lecture d'un entier dans la variable entière `n`;
`scanf("%f", &x)`	:	lecture d'un réel dans la variable flottante `x`;

1.7.4 Variables globales

Nous avons pour l'instant considéré seulement deux familles de variables de natures très voisines, les paramètres formels de fonctions et les variables locales

à un bloc. Il en existe une troisième, les variables déclarées directement au niveau zéro dans le fichier, c'est-à-dire hors de tout bloc.

De façon similaire aux déclarations de fonctions, ces variables sont visibles dans tout le fichier. Cela implique qu'elles *"existent"* indépendamment de l'exécution de toute fonction. Elles sont initialisables, mais cette initialisation est effectuée une fois et une seule, lors de la compilation. Elles permettent de partager des données entre plusieurs fonctions d'un même fichier.

Si deux variables de niveau zéro situées dans deux fichiers différents portent le même nom, elles désignent la même zone mémoire. Cela signifie que le partage de données peut s'effectuer entre des fonctions quelconques du programme, éventuellement situées dans des fichiers différents. Ces variables de niveaux zéro constituent l'**environnement global** du programme.

Considérons le fichier `main_divi.c` suivant :

main_divi.c

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

void usage();
extern float division(float, float);

char *argv0;

main(int argc, char *argv[])
{
    float f1;
    float f2;

    argv0 = *argv;
    if (argc < 3)
        usage();
    f1 = atof(argv[1]);
    f2 = atof(argv[2]);
    printf("%08.2f\n-------- = %08.2f\n%08.2f\n",
           f1, division(f1, f2), f2);
}

void usage()
{
    fprintf(stderr, "Usage: %s reel reel\n", argv0);
    exit(1);
}
```

main_divi.c

Le fichier `main_divi.c` fait partie du programme `divi` imprimant le quotient de deux nombres réels. La variable `argv0` est une variable globale du programme. Elle est utilisée au début de la fonction `main` lors de son initialisation par une affectation, et dans la fonction `usage` lors de l'impression d'un message d'erreur correspondant à une invocation incorrecte de la commande.

Cette erreur est détectée en testant le nombre d'arguments de la commande au moyen de la structure de contrôle de test `if`, sur laquelle nous reviendrons dans la section suivante. L'impression du message d'erreur est effectuée au moyen de la fonction `fprintf`. C'est une généralisation de la fonction `printf`

permettant d'indiquer la sortie sur laquelle on désire écrire. La fonction `printf` est associée à une sortie par défaut, la sortie standard, encore appelée `stdout`. Pour l'affichage des messages d'erreurs, on utilise de préférence une autre sortie appelée `stderr`, pour *standard erreur*. Les définitions de `stdout` et `stderr` sont contenues dans `<stdio.h>`. La fonction `exit` appelée ensuite provoque la terminaison du programme; la valeur 1 du paramètre représente un code d'erreur. Son prototype est défini dans `<stdlib.h>`. Le fichier `divi.c` suivant contient la fonction `division` effectuant le calcul du quotient :

```
═══════ divi.c ═══════════════════════════
#include <stdio.h>
#include <stdlib.h>

extern char *argv0;

float division(float n, float d)
{
    if (d == 0)
    {
        fprintf(stderr, "\"%s\": division par zero\n", argv0);
        exit(1);
    }
    return n/d;
}
═══════════════════════════ divi.c ═══════
```

La variable `argv0` fait référence à la variable globale de même nom définie dans le fichier `main_divi.c`. Sa valeur est la chaîne de caractères codant le nom de la commande. Voici les trois scénarios possibles d'exécution de ce programme; on peut vérifier que la variable `argv0` est commune à tout le programme.

```
$ divi
Usage: divi reel reel
$
$ divi 22 7
00022.00
-------- = 00003.14
00007.00
$
$ divi 1 0
"divi": division par zero
$
```

Ici encore, après un renommage de la commande `divi`, les messages d'erreur seront identifiés par le nouveau nom.

Il existe enfin la possibilité de définir des variables et des fonctions visibles depuis toutes les fonctions d'un fichier, mais locales à ce fichier. Les variables locales d'un fichier constituent son **environnement local**. Une déclaration est rendue locale au fichier qui la contient au moyen du mot-clé `static`. Celui-ci se place en tête de la déclaration :

```
static void usage(char *);
```

Ces premiers pas dans le monde des programmes C nous ont permis de nous familiariser avec le concept de fonction qui en est l'élément clé. Nous avons également découvert les expressions qui sont les constructions élémentaires sur lesquelles repose toute l'architecture. Les outils de base sont en place. Nous avons maintenant besoin de structures de contrôle pour former des instructions plus puissantes que les instructions élémentaires avec lesquelles nous avons fait nos premiers essais. C'est ce que nous allons aborder dans la section suivante.

1.8 Tests et itérations

1.8.1 Test : `if-else`

Nous allons modifier le programme `delta` calculant le discriminant d'une équation du second degré, pour construire le programme `racines` effectuant le calcul des racines de l'équation. La première instruction de ce programme est une instruction de test de la forme :

```
if (⟨expression⟩)
    ⟨instruction⟩
```

On trouve, après quatre affectations, un nouveau test de la forme :

```
if (⟨expression⟩)
    ⟨instruction⟩
else
    ⟨bloc⟩
```

Plus généralement, il peut y avoir après la ligne du `if` et après celle du `else` indifféremment une instruction ou un bloc. Ce sont respectivement la *partie alors* et la *partie sinon* du test. La partie sinon peut être omise, comme c'est le cas pour le premier test de l'exemple. Cette construction peut s'appliquer récursivement : le bloc de la *partie sinon* du second test est lui-même une instruction de test. Suivant la valeur de l'expression, c'est la *partie alors* ou la *partie sinon* qui est exécutée.

racines.c

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

static void usage(char *s);

main(int argc, char *argv[])
{
    double a;
    double b;
    double c;
    double delta;

    if (argc != 4)
        usage(argv[0]);

    a = atof(argv[1]);
```

```
    b = atof(argv[2]);
    c = atof(argv[3]);
    delta = b*b - 4*a*c;

    if (delta < 0)
        printf("Pas de racine reelle.\n");
    else
    {
        if (delta == 0)
            printf("Une racine: %g\n", -b/(2*a));
        else
        {
            double racdelta = sqrt(delta);

            printf("Deux racines: %g et %g\n",
                   (-b - racdelta)/(2*a),
                   (-b + racdelta)/(2*a));
        }
    }
}

static void usage(char *s)
{
    fprintf(stderr, "Usage: %s a b c\n", s);
    exit(1);
}
```

racines.c

L'instruction

```
if (argc != 4)
    usage(argv[0]);
...
```

au début du programme permet de détecter une utilisation incorrecte de la commande. On obtiendra de cette façon, lors d'un lancement de `racines` sans argument, le message :

```
$ racines
Usage: racines a b c
$
```

à la place d'un message d'erreur comme :

```
$ racines
Segmentation fault (core dumped)
$
```

provoqué ici par un appel à la fonction `atof` avec un paramètre invalide.

Ce programme utilise la fonction de bibliothèque `sqrt` calculant la racine carrée d'un réel double précision. Son prototype est défini dans le fichier en-tête `<math.h>`.

Sous UNIX, cette fonction, comme toutes les fonctions mathématiques de la bibliothèque standard, n'est pas chargée automatiquement par l'éditeur de liens lorsqu'il construit un exécutable. La compilation du programme `racine` par la commande

```
gcc -g racine.c -o racine
```

provoque un message d'erreur, émis par l'éditeur de liens, signalant que le symbole sqrt n'est pas défini :

```
$ gcc -g racine.c -o racine
ld: Undefined symbol
   _sqrt
$
```

Remarque 5 *Sous* UNIX*, le caractère _ est rajouté devant chaque symbole global du programme par l'éditeur de liens. Cela permet de réserver les identificateurs ne commençant pas par ce caractère pour des fonctions internes à l'implémentation (voir 12.1.4).*

L'option -l permet d'indiquer des bibliothèques spécifiques à l'éditeur de liens. Il suffit de concaténer le nom de la bibliothèque à la suite de la lettre l, dans notre cas tout simplement le nom m :

```
$ gcc -g racine.c -o racines -lm
$
$ racines 2 5 3
Deux racines: -1.5 et -1
$ racines 1 2 1
Une racine: -1
$ racines 1 1 1
Pas de racine relle.
$
```

La construction du programme précédent peut s'écrire un peu plus simplement :

```
...
if (delta < 0)
    printf("Pas de racine relle.\n");
else if (delta == 0)
    printf("Une racine: %f\n", -b/(2*a));
else {
    double sqrt();
    double racdelta = sqrt(delta);

    printf("Deux racines: %f et %f\n",
           (-b - racdelta)/(2*a),
           (-b + racdelta)/(2*a));
}
...
```

En effet lorsque plusieurs tests sont enchaînés, une *partie sinon* se rapporte à la *partie alors* libre la plus proche.

1.8.2 Itération : while

L'instruction d'itération la plus simple est l'instruction **while** répétant l'exécution d'une instruction ou d'un bloc d'instructions tant que la valeur d'une expression n'est pas nulle. Ceci est illustré par la fonction **affiche_cars** suivante :

```
═══════ affiche_cars.c ═══════════════════════════
#include "affiche_cars.h"

void affiche_cars(char c, int n)
{
    while (n > 0)
    {
        putchar(c);
        n--;
    }
}
═══════════════════════════ affiche_cars.c ═══════
```

Cette fonction répète *n* fois l'affichage du caractère *c*. On remarquera l'expression à effet de bord `n--` qui décrémente de 1 le contenu de la variable `n`. La valeur de cette expression est la valeur de `n` avant décrémentation; on peut donc également écrire cette boucle

```
...
while (n-- > 0)
    putchar(c);
...
```

Afin de ne pas avoir à retaper le prototype de la fonction **affiche_cars** on le place un fichier en-tête `affiche_cars.h` associé au fichier `affiche_cars.c`.

```
═══════ affiche_cars.h ═══════════════════════════
#ifndef AFFICHE_CARS_H
#define AFFICHE_CARS_H

extern void affiche_cars(char, int);

#endif  /* AFFICHE_CARS_H */
═══════════════════════════ affiche_cars.h ═══════
```

Nous reviendrons plus tard sur l'utilisation des directives **#ifdef**, **#define** et **#endif**. Ce fichier, à la différence des fichiers en-tête de l'environnement standard, est placé dans le répertoire courant. On le signale au préprocesseur en plaçant son nom entre guillemets (au lieu des chevrons utilisés pour les fichiers en-tête standard) :

```
#include "affiche_cars.h"
```

On notera qu'il est également inclus dans le fichier `.c` associé.

Voici un exemple d'utilisation de la fonction `afficher` pour écrire un programme réalisant l'encadrement d'une chaîne de caractères transmise en argument.

encadre.c

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

#include "affiche_cars.h"

static void encadre(char, char *);
static void usage(char *);

main(int argc, char *argv[])
{
    if (argc == 3)
        encadre(argv[1][0], argv[2]);
    else
        usage(argv[0]);
}

static void encadre(char c, char *s)
{
    int longueur = strlen(s);

    printf("\n  ");
    affiche_cars(c, longueur+4);
    printf("\n  %c %s %c\n  ", c, s, c);
    affiche_cars(c, longueur+4);
    printf("\n\n");
}

static void usage(char *s)
{
    fprintf(stderr, "Usage: %s caractere chaine\n", s);
    exit(1);
}
```

encadre.c

On remarquera dans ce programme l'expression `argv[1][0]`. Nous savons que l'expression `argv[1]` est une chaîne de caractères; or une chaîne est aussi un vecteur de caractères, dont les éléments sont accessibles séparément. L'expression `argv[1][0]` permet par conséquent d'isoler le premier caractère de la chaîne `argv[1]`. Nous reviendrons plus loin sur l'utilisation du mot clé `static`.

On remarquera également la fonction `strlen` qui retourne la longueur d'une chaîne passée en paramètre. La commande `encadre` s'utilise de la façon suivante :

```
$ encadre # "Une phrase encadree se voit mieux"

  #####################################
  # Une phrase encadree se voit mieux #
  #####################################

$ encadre + "`date`"

  ++++++++++++++++++++++++++++++++
  + Tue Dec 26 17:10:08 MET 1989 +
  ++++++++++++++++++++++++++++++++
```

```
$
```

Le second exemple utilise le mécanisme shell de substitution de commande.

Un exemple incontournable d'utilisation de la boucle *tant que* est celui du programme comptant le nombre de caractères lus sur son entrée standard :

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

main()
{
   int nombre_car = 0;

   while (getchar() != EOF)
      nombre_car++;
   printf("%d caractere%s\n", nombre_car,
         (nombre_car > 1) ? "s" : "");
}
taille.c
```

La fonction getchar fait partie de la bibliothèque standard; elle renvoie un caractère lu sur l'entrée standard. Sur l'exemple suivant, la première exécution s'arrête avec la frappe du caractère CONTROL-D, également noté ^D ou C-d, qui signifie *fin de fichier* sous UNIX. Le second exemple utilise une redirection.

```
$ taille
abcdef
gh
C-d
10 caracteres
$
$ taille < taille.c
186 caracteres
$
```

Le caractère EOF, utilisé pour détecter la fin de fichier en lecture, est défini dans le fichier en-tête <stdio.h>. On peut définir de nouvelles constantes au moyen de la directive define. Par exemple

```
#define TAILLE_BLOC 1024
```

définit le symbole TAILLE_BLOC comme étant la constante entière 1024 et peut être utilisé à sa place, n'importe où dans le fichier :

```
char buffer[TAILLE_BLOC];
:
while (i < TAILLE_BLOC)
    :
```

1.8.3 Itération : `for`

Une autre structure de contrôle très utilisée est la boucle `for` dont le déroulement est contrôlé par trois expressions :

1) une expression d'initialisation exécutée une fois au début de l'exécution de la boucle;
2) une expression conditionnant l'arrêt de la boucle, exécutée avant chaque itération;
3) une expression exécutée à la fin de chaque itération et préparant l'itération suivante.

Ces trois expressions sont placées entre parenthèses et séparées par un point-virgule. Par exemple l'instruction

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

répète n fois l'exécution de ⟨*instruction*⟩, successivement pour les valeurs 0, 1, ..., $n-1$ de `i`. Comme précédemment, le corps de boucle peut être une instruction ou un bloc.

Il existe un opérateur d'expression permettant de construire des **listes d'expressions**. Il s'agit de l'opérateur

`,`

concaténant deux expressions quelconques. Dans l'exemple qui suit, la troisième expression de l'instruction `for` est construite avec cet opérateur.

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

#include "affiche_cars.h"

#define MOTIF '^'

static void pyra(int);
static void usage(char*);

main(int argc, char *argv[])
{
    if (argc != 2)
        usage(argv[0]);
    pyra(atoi(argv[1]));
}

static void pyra(int hauteur)
{
    int nblancs = hauteur-1;
    int i;

    for (i = 0; i < hauteur; nblancs--, i++)
    {
        affiche_cars(' ', nblancs);
        affiche_cars(MOTIF, 2*i+1);
        printf("\n");
    }
```

```
}

static void usage(char *s)
{
    printf("Usage: %s hauteur\n", s);
    exit(1);
}
```
═══════════════ pyra.c ═══════

Ce programme affiche une pyramide dont la hauteur est passée en argument :

```
$ pyra
Usage: pyra hauteur
$ pyra 5
    ^

   ^^^

  ^^^^^

 ^^^^^^^

^^^^^^^^^

$
```

La structure de contrôle `for` peut être utilisée de nombreuses façons différentes. Parfois, le traitement de l'itération est entièrement contenu dans les trois expressions de contrôle; dans ce cas, le corps de boucle est une instruction vide. Par exemple, l'instruction

```
for (nb = 0; getchar() != EOF; nb++)
    ;
```

compte le nombre de caractères lus sur l'entrée standard. De même, l'instruction

```
for (i = 1; i < argc; i++)
    traiter(argv[i]);
```

placée dans la fonction `main` itère un traitement sur tous les arguments d'une commande, excepté le nom de cette commande, le paramètre `argv[0]` (voir section 1.6 page 23). Voici un exemple illustrant la récupération des arguments d'une commande.

═══════ arguments.c ═══════════════

```
#include <stdio.h>

main(int argc, char *argv[])
{
    if (argc == 1)
        printf("Rien a faire.\n");
    else if (argc == 2)
        printf("Traitement de l'argument %s.\n", argv[1]);
    else
    {
        int i;

        printf("Traitement des arguments");
        for (i = 1; i < argc-2; i++)
            printf(" %s,", argv[i]);
```

```
        printf(" %s et %s.\n", argv[argc-2], argv[argc-1]);
    }
}
```
arguments.c

```
$ arguments
Rien a faire.
$
$ arguments un
Traitement de l'argument un.
$
$ arguments un deux
Traitement des arguments un et deux.
$
$ arguments un deux trois quatre cinq
Traitement des arguments un, deux, trois, quatre et cinq.
$
```

Il est intéressant de remarquer que ce mécanisme de récupération d'arguments est parfaitement adapté aux interprètes de commande UNIX gérant des listes de mots. En particulier, le programmeur bénéficie des mécanismes de complétion de l'interprète, comme par exemple les substitutions de noms de fichiers :

```
$ echo *.c
arguments.c pyra.c taille.c
$ arguments *.c
Traitement des arguments arguments.c, pyra.c et taille.c.
$
```

Sur cet exemple, la liste des arguments est le résultat de la substitution du motif `*.c` par la liste de tous les fichiers du répertoire courant dont le nom se termine par `.c`. Il en résulte deux conséquences fondamentales :

- pas de limitation théorique sur le nombre d'arguments d'une commande;
- homogénéité de comportement de toutes les commandes, que ce soient des commandes standard ou des programmes utilisateur.

1.9 Quelques règles de présentation

Pour écrire des programmes C lisibles, il est fondamental de respecter un certain nombre de règles de présentation. Ces règles portent sur le découpage du programme en lignes, et l'**indentation** de chaque ligne, c'est-à-dire son décalage par rapport à la marge gauche.

La règle stricte à respecter pour effectuer le découpage est de ne jamais placer plusieurs instructions sur la même ligne. L'indentation doit, quant à elle, rendre compte fidèlement de la structure syntaxique du programme.

Certains éditeurs de texte sont capables d'effectuer automatiquement l'indentation des lignes. Le plus performant est actuellement l'éditeur *GNU Emacs* de Richard Stallman. Cet éditeur fait partie du projet *GNU* (voir la note de la page 11).

Il existe également des commandes de formatage, comme par exemple la commande **CB(1)**. Dans tous les cas, pour que ces aides à la programmation opèrent correctement, il convient d'effectuer convenablement le découpage en lignes du programme.

1.9.1 Découpage en lignes

Le principe général régissant les règles de découpage est d'isoler chaque élément structurel sur une ligne : instructions élémentaires, délimiteurs de bloc, structures de contrôle, etc.

Règle 1. Une déclaration commence et termine sa ligne

Par conséquent, le point virgule d'une déclaration est toujours le dernier caractère de la ligne :

```
int x; int y=1;        int x;
                       int y=1;
```

Règle 2. Une instruction élémentaire commence et termine sa ligne.

Comme dans le cas des déclarations, le point virgule d'une instruction est toujours le dernier caractère de la ligne :

```
i = 0; j = 0;          i = 0;
                       j = 0;
```

De même, l'instruction d'une structure de contrôle commence sur une nouvelle ligne, même lorsqu'il s'agit de l'instruction vide, c'est-à-dire réduite à un point-virgule.

```
while (*p++);          if (x<0) x=0;

while (*p++)           if (x<0)
    ;                      x=0;
```

Règle 3. Une structure de contrôle commence sa ligne.

Cas particulier : la forme `else if` est considérée ici comme une structure de contrôle à part entière; on écrira donc

```
else if (...
```

et non

```
else
    if (...
```

Règle 4. Une accolade ouvrante termine sa ligne.

L'accolade ouvrante du début d'un bloc peut être placée à la fin d'une ligne, ou bien seule sur la ligne suivante; dans le second cas, on trouve deux styles d'indentation différents.

```
if (x>y) { t=x; x=y; y=t; }

if (x>y) {        if (x>y)          if (x>y)
    t=x;          {                   {
    x=y;              t=x;              t=x;
    y=t;              x=y;              x=y;
}                     y=t;              y=t;
                  }                   }
```

Règle 5. Une accolade fermante est seule sur sa ligne.

```
}}        }
         }
```

Règle 6. Une étiquette commence et termine sa ligne.

Cette règle s'applique également à l'étiquette d'un champ d'aiguillage :

```
erreur:free(p);          switch(c) {
        return 0;          case 'a': append();
                                     break;
                           ...
```

```
erreur:                switch(c) {      switch(c)
  free(p);               case 'a':      {
  return 0;                append();      case 'a':
                           break;           append();
                         ...                break;
                                          ...
```

Enfin, il est d'usage d'insérer une ligne blanche après la dernière déclaration d'une liste de déclarations. Voici un programme C dans lequel les découpages sont effectués indépendamment de sa structure syntaxique :

```
subsep_en_vrac.c
#include <stdio.h>
#include <string.h>
main(int argc, char *argv[]) { if (argc != 2) fprintf(stderr,
"Usage: %s <mot>\n", argv[0]); else {int i; for (i = 0; i <
strlen(argv[1]); i++) { char c = argv[1][i]; switch (c) {
case '-':  case '_':putchar(' '); break; default: putchar(c);
break; } } printf("\n"); } }
subsep_en_vrac.c
```

Si on applique les règles **1** à **6** que nous venons d'énoncer, un découpage correct de ce programme est le suivant :

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

main (int argc, char *argv[])
{
if (argc != 2)
fprintf(stderr, "Usage: %s <mot>\n", argv[0]);
else
{
int i;

for (i = 0; i < strlen(argv[1]); i++)
{
char c = argv[1][i];

switch (c)
{
case '-':
```

```
case '_':
putchar(' ');
break;
default:
putchar(c);
break;
}
}
printf("\n");
}
}
```

subsep_decoupe.c

1.9.2 Indentation

Une indentation correcte est indispensable à la lecture et la compréhension d'un programme, d'où l'importance d'utiliser un éditeur capable d'effectuer automatiquement cette indentation.

L'indentation de chaque instruction dépend de son niveau de profondeur dans le programme. Les décalages doivent être constants. Les valeurs de décalage les plus couramment utilisées sont de deux, quatre, ou huit caractères.

Le décalage s'effectue vers la droite sur la première instruction d'un bloc et sur l'instruction d'une structure de contrôle. Si une accolade ouvrante est seule sur sa ligne, elle peut éventuellement faire l'objet d'un demi-décalage vers la droite.

Les étiquettes peuvent faire l'objet de décalages spécifiques. Sur les exemples présentés dans cet ouvrage, les lignes contenant une étiquette sont décalées vers la gauche d'un nombre de colonnes égal à la moitié de la valeur de décalage. La valeur de décalage adoptée dans cet ouvrage étant de quatre colonnes, les étiquettes sont décalées de deux colonnes vers la gauche.

Une accolade fermante est placée sur la même colonne que l'accolade ouvrante associée. Si l'accolade ouvrante du bloc principal d'une fonction est placée sur la première colonne du texte, et si l'indentation de la fonction est correctement effectuée, l'accolade fermante doit également se trouver en première colonne. Si cela n'est pas le cas, le source de la fonction comporte des erreurs de syntaxe.

Ces règles, communément admises par la communauté des programmeurs C, sont également celles utilisées par l'éditeur *GNU Emacs*. Voici, par exemple, le résultat de l'indentation calculée automatiquement par *GNU Emacs* sur le résultat du découpage précédent :

subsep_indent_1.c

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

main (int argc, char *argv[])
{
  if (argc != 2)
    fprintf(stderr, "Usage: %s <mot>\n", argv[0]);
  else
```

```
    {
      int i;

      for (i = 0; i < strlen(argv[1]); i++)
        {
          char c = argv[1][i];

          switch (c)
            {
            case '-':
            case '_':
              putchar(' ');
              break;
            default:
              putchar(c);
              break;
            }
        }
      printf("\n");
    }
}
```

subsep_indent_1.c

Dans cet ouvrage, nous avons adopté un style d'indentation légèrement différent :

subsep_indent_2.c

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

main (int argc, char *argv[])
{
    if (argc != 2)
        fprintf(stderr, "Usage: %s <mot>\n", argv[0]);
    else
    {
        int i;

        for (i = 0; i < strlen(argv[1]); i++)
        {
            char c = argv[1][i];

            switch (c)
            {
              case '-':
              case '_':
                putchar(' ');
                break;
              default:
                putchar(c);
                break;
            }
        }
        printf("\n");
    }
}
```

═══════════ subsep_indent_2.c ═══════

Ce style est obtenu en paramétrant, l'éditeur *GNU Emacs*. À titre d'information, voici le paramétrage utilisé :

```
(setq c-indent-level 4)
(setq c-continued-statement-offset 4)
(setq c-continued-brace-offset -4)
(setq c-brace-offset 0)
(setq c-argdecl-indent 4)
(setq c-label-offset -2)
```

1.10 GNU Emacs et le langage C

Le but de cette section n'est pas de familiariser le lecteur avec *GNU Emacs*, mais seulement de donner les principales fonctionnalités de l'éditeur concernant l'édition et la mise au point d'un programme en langage C.

Le comportement de l'éditeur *GNU Emacs* est paramétré par le type du fichier manipulé. Plusieurs **modes** sont prédéfinis, dont le mode *langage* C ou `c-mode`. Il est, par défaut, activé automatiquement lors de l'édition d'un fichier suffixé `.c` ou `.h`.

Les commandes de *GNU Emacs* sont activées au moyen de **clés** qui sont des combinaisons de touches du clavier. Par exemple la clé `C-n`[10] est formée de la combinaison des touches `CONTROL` et `n` enfoncées simultanément. La clé `M-x` correspond à la combinaison de touches `META` et `x` si le clavier possède une touche `META`, et à la suite de touches `ESCAPE x` sinon. Enfin, la clé `C-M-p` est composée de la combinaison des touches `CONTROL`, `META`, et `p` enfoncées simultanément, ou de la touche `ESCAPE` suivie de la combinaison `CONTROL-p`.

Par défaut, *GNU Emacs* effectue divers contrôles, comme par exemple celui de la validité du parenthésage. Il est possible de déplacer le curseur de blocs de parenthèses en blocs de parenthèses au moyen des clés `C-M-p` et `C-M-n`.

1.10.1 Marquage de régions et indentation

La commande

`indent-region`

de *GNU Emacs* permet d'effectuer l'indentation de la région du texte comprise entre la position courante du curseur et une position préalablement définie appelée la **marque**. Il est possible de positionner la marque sur la position courante du curseur en tapant la clé `C-@` ou `C-ESPACE`.

On peut faire exécuter la commande `indent-region` de plusieurs façons :

– `M-x indent-region` : la commande `M-x` permet d'entrer en mode commande; le nom de la commande suivi de RETURN déclenche son exécution;

[10] Les deux notations C-*x* et ^*x* sont couramment utilisées pour représenter le caractère `CONTROL-`*x*.

- `C-M-\` : la commande `indent-region` est liée à la clé `C-M-\` et peut être invoquée directement de cette façon.

Il est possible de définir la région courante comme étant la fonction C dans laquelle est positionné le curseur au moyen de la commande

```
mark-c-function
```

La marque est placée à la fin de la fonction et le curseur est amené au début. Par défaut, cette commande n'est liée à aucune clé.

L'indentation de la ligne courante peut être effectuée en tapant tout simplement la clé `TAB`. L'indentation est calculée par rapport à celle des lignes précédentes.

Quel que soit le style d'indentation adopté, nous insistons encore une fois sur le fait qu'il est **fondamental** que ce style soit, d'une part cohérent avec la structure syntaxique du langage, et d'autre part strictement appliqué lors de toute opération de codage en C. **Une indentation correcte peut permettre de déceler des erreurs d'ordre sémantique.** Considérons par exemple la fonction

```
copier(int argc, char *argv[])
{
    int i;

    for (i=0; i < argc; i++);
        copier(argv[i]);
}
```

Lors de son exécution, cette fonction ne produit strictement aucun résultat, sauf peut-être une sortie sur erreur, et ceci quelles que soient les valeurs de `argc` et `argv`. Indentée par *Emacs*, son source est modifié en

```
copier(int argc, char *argv[])
{
    int i;

    for (i=0; i < argc; i++);
    copier(argv[i]);
}
```

Il devient alors évident que l'instruction `copier(argv[i]);` ne fait pas partie de la boucle `for`. En poussant l'examen un peu plus loin, on s'aperçoit qu'un point virgule indésirable s'est glissé à la suite de la parenthèse de la boucle `for`, faisant en sorte que le corps de boucle devienne une instruction vide. Il s'agit là non d'une erreur syntaxique mais bien d'une erreur sémantique indétectable par le compilateur.

1.10.2 Compilation et correction d'erreurs

La commande

```
compile
```

permet de lancer la compilation d'un programme directement depuis une fenêtre de l'éditeur; les messages d'erreur éventuels sont alors affichés dans cette même fenêtre. Par défaut, *Emacs* invoque la commande `make`; il est possible d'entrer une commande quelconque.

Cette manière de procéder offre deux avantages :

1) avant de lancer la compilation, *Emacs* recherche parmi ses *buffers* s'il en existe qui ont été modifiés depuis leur dernière écriture sur disque; s'il trouve de tels buffers, il propose de les sauvegarder, ce qui évite de recompiler un fichier qu'on aurait oublié de sauvegarder;
2) lorsqu'il y a des erreurs de syntaxe, la clé

   `C-x ‘`

   permet de se positionner automatiquement dans le fichier contenant la première erreur, et sur la ligne de l'erreur; chaque nouvelle frappe de la clé `C-x ‘` provoque le positionnement sur l'erreur suivante, et ceci jusqu'à ce que toutes les erreurs aient été défilées.

On peut voir l'illustration de ce fonctionnement sur la figure 1.3. On a lancé la compilation du fichier `subsep_indent_2.c` dans lequel ont été introduites deux erreurs de syntaxe. Dans la fenêtre supérieure, le curseur est positionné sur la première ligne erronée; le message d'erreur correspondant est affiché sur la première ligne de la fenêtre inférieure. Il est possible de corriger directement la faute, puis de se positionner sur l'erreur suivante en tappant `C-x ‘`. Cela provoquera un défilement du contenu de la fenêtre inférieure, de façon à amener la ligne `subsep_indent_2.c:24: parse...` sur la première ligne. Dans la fenêtre supérieure, le curseur est alors positionné sur la ligne 24.

```
emacs-19.15@nemo
Buffers   File   Edit   Help
            char c = argv[1][i];

            switch (c)
            {
              case '-':
              case '_':
                put char(' ');
                break;
              default:
                putchar(c)
                break;
            }
-----Emacs: subsep_indent_2.c        (C)--44%-----------------------------
subsep_indent_2.c:20: 'put' undeclared (first use this function)
subsep_indent_2.c:20: (Each undeclared identifier is reported only once
subsep_indent_2.c:20: for each function it appears in.)
subsep_indent_2.c:20: parse error before 'char'
subsep_indent_2.c:24: parse error before 'break'

Compilation exited abnormally with code 1 at Thu Jul 15 11:19:10




--%%-Emacs: *compilation*        (Compilation: exit)--Bot-------------------
```

Figure 1.3: Exemple de compilation sous emacs

Signalons enfin la commande

`c-macro-expand`

qui réalise, en invoquant le préprocesseur C, l'expansion des pseudo-fonctions et les inclusions de fichiers sur la région courante.

Chapitre 2

Les déclarations

Sous sa forme la plus simple, une déclaration est composée d'un type suivi d'une liste de noms d'objet. Par exemple,

```
int    Debut, Milieu, Fin;
```

déclare les trois variables entières `Debut`, `Milieu` et `Fin`. Il est possible de construire des objets plus complexes en appliquant des constructeurs sur un nom d'objet. Par exemple, la déclaration suivante utilise le constructeur `[]` pour déclarer un vecteur de sept entiers, de nom `total_par_jour` :

```
int total_par_jour[7];
```

Chaque objet est caractérisé par son type, mais aussi par son **implémentation** et sa **visibilité**. L'implémentation d'un objet concerne principalement sa durée de vie : il peut être alloué dynamiquement lors de l'exécution du programme ou statiquement par le compilateur. Sa visibilité détermine les parties du programme dans lesquelles il est accessible.

2.1 Types de base et types élémentaires

2.1.1 Types de base

Les types de base sont les caractères, les entiers et les réels simple et double précision, identifiés par les mots-clés : `char`, `int`, `float`, et `double`. Il existe également un type *ensemble vide* ne contenant aucune valeur : le type `void`. Il est, entre autres, utilisé

- pour déclarer des procédures, c'est-à-dire de fonctions ne retournant aucune valeur;
- pour écrire le prototype d'une fonction sans paramètres :

```
int getchar(void)
```

- pour écarter explicitement la valeur d'une expression :

```
(void) getchar();
```

Nous verrons également une autre construction, effectuée avec ce type, appelée *pointeur générique*.

Les mots-clés `short` et `long` permettent d'influer sur la taille mémoire des types de base. Le tableau 2.1 donne la liste de tous les types de base qu'il est

ainsi possible de former.

——	`void`	*ensemble vide*
`unsigned`	`char`	caractère
(sans signe)	`short int`	entier court
`signed`	`int`	entier par défaut
(signé)	`long int`	entier long
	`float`	réel simple précision
——	`double`	réel double précision
	`long double`	réel précision étendue

Tableau 2.1: Types de base

Sur les machines dont la mémoire est organisée en octets de 8 bits, le type `char` codant un caractère est également implémenté sur 8 bits. Il correspond par conséquent à la plus petite unité de mémoire adressable.

La taille d'un entier par défaut correspond le plus souvent à la taille d'un mot machine. Les **entiers courts** sont généralement codés sur deux octets, et les **entiers longs** sur quatre.

Les types `short int` et `long int` peuvent être abrégés en `short` et `long`. La taille d'un objet de type `int` peut varier selon les machines, et éventuellement les versions de compilateur. On déclarera une variable de type `int` chaque fois qu'on ne fait pas d'hypothèse sur sa taille. On optimise de cette façon les accès mémoire à cette variable, car ils correspondent en général au mode d'adressage par défaut de la machine. Dans le cas contraire, on utilisera les types explicites `short` ou `long`.

Dans un souci de portabilité, on ne fera aucune supposition sur la taille en octets d'un objet de type `int`.

Par défaut, un entier est une quantité **signée**, c'est-à-dire pouvant prendre des valeurs positives et négatives. Un entier codé sur deux octets est compris entre -2^{15} et $2^{15}-1$, c'est-à-dire -32768 et 32767. Un entier codé sur quatre octets est compris entre -2^{31} et $2^{31}-1$, c'est-à-dire -2147483648 et 2147483647. Un entier est dit **sans signe** lorsque qu'il ne prend que des valeurs positives ou nulles. Un entier sans signe codé sur deux octets est compris entre 0 et $2^{16}-1$ (c'est-à-dire 65535); un entier sans signe codé sur quatre octets est compris entre 0 et $2^{32}-1$ (c'est-à-dire 4294967295).

Un caractère peut être une quantité signée ou sans signe, mais à la différence des entiers, cela peut varier selon les implémentations. En général, un caractère est signé et peut prendre des valeurs entre -128 et 127, mais un programme dans lequel cette hypothèse est utilisée n'est pas portable.

On peut spécifier qu'un caractère ou un entier est sans signe en préfixant sa déclaration par le mot-clé `unsigned`. Par exemple,

```
unsigned short niveau;
```

déclare un entier court sans signe. Le programme suivant illustre quelques différences entre l'arithmétique non signée et signée.

unsigned.c

```
#include <stdio.h>

main()
{
    short sh;
    unsigned short u_sh;

    sh = u_sh = 0x7fff;
    printf("Valeurs initiales\n");
    printf("  sh   = %d (0x%x)\n", sh, sh);
    printf("  u_sh = %d (0x%x)\n", u_sh, u_sh);

    sh++, u_sh++;
    printf("\nValeurs apres incrementation\n");
    printf("  sh   = %d (0x%x)\n", sh, sh);
    printf("  u_sh = %d (0x%x)\n", u_sh, u_sh);

    u_sh = 0xffff;
    printf("\nValeur max d'entier court sans signe: %d\n",
           u_sh);
    u_sh++;
    printf("Valeur suivante: %d\n", u_sh);
    printf("Valeur signee de 0xffffffff: %d\n",
           0xffffffff);
    printf("Valeur sans signe de 0xffffffff: %u\n",
           0xffffffff);
}
```

unsigned.c

Les deux variables sh et u_sh, de type `short` et `unsigned short`, sont initialisées avec la valeur 32767, qui s'écrit `0x7fff` en hexadécimal (les 15 bits de poids faible sont positionnés à 1); elles sont ensuite incrémentées de 1. La valeur initiale étant la valeur maximale d'un entier court signé, le résultat est différent pour les deux variables. Les dernières lignes du programme font afficher les valeurs maximales des autres types entiers de base.

```
$ unsigned
Valeurs initiales
  sh   = 32767 (0x7fff)
  u_sh = 32767 (0x7fff)
Valeurs apres incrementation
  sh   = -32768 (0xffff8000)
  u_sh = 32768 (0x8000)
Valeur max d'entier court sans signe: 65535
Valeur suivante: 0
Valeur signee de 0xffffffff: -1
Valeur sans signe de 0xffffffff: 4294967295
$
```

Remarque 6 *L'incrémentation de* sh *et de* u_sh *produit le même mot de 16 bits, à savoir la valeur 0x8000. Lors de l'appel à la fonction* `printf`*, ces valeurs sont converties en entiers longs avant d'être passées en paramètres; la valeur*

contenue dans sh *étant considérée comme un nombre négatif, il y a extension du bit de signe.*

Il est possible de spécifier explicitement qu'un entier ou un caractère est signé au moyen du mot-clé **signed**. Par exemple, la déclaration d'un caractère signé s'écrit :

```
signed char c;
```

Les types **float** et **double** sont en général implémentés sur respectivement, deux ou quatre octets, et quatre ou huit octets. Le type **long double** a été ajouté en C ANSI pour permettre de spécifier, lorsqu'ils existent, des réels en précision étendue, codés sur seize octets.

2.1.2 Types élémentaires

Les **types élémentaires** sont les types de base, les pointeurs, et les énumérations de constantes. La taille d'un **objet élémentaire** (c'est-à-dire un objet de type élémentaire) est fixée, contrairement aux vecteurs par exemple dont la taille dépend de chaque construction. Cette taille varie en général d'un octet pour les caractères, à huit octets pour les réels double précision, voire plus pour les réels précision étendue.

Enumération de constantes

Une énumération de constantes, ou plus simplement **énumération**, est une construction définissant un type à partir une liste explicite de valeurs symboliques. Les valeurs sont définies au moyen de symboles de constante. La syntaxe d'une énumération est :

enum ⟨*type*⟩ { ⟨*identificateur*⟩ [= ⟨*expression-constante*⟩] [,...]}

Par exemple

```
enum Feu { Vert, Orange, Rouge, Orange_clignotant, En_panne }
```

définit le type **enum Feu** comme l'ensemble des cinq constantes

```
Vert
  ⋮
En_panne
```

Chaque constante est codée par une valeur entière. Sur l'exemple, **Vert** a pour valeur zéro, et **En_panne** a pour valeur quatre. Il est possible de spécifier explicitement la valeur des constantes. Par exemple

```
enum Operateur
{
    Plus = '+',
    Moins = '-',
    Multiplie = '*',
    Divise = '/',
    Modulo = '%'
};
```

définit le type **Operateur** comme la liste des symboles **Plus**, **Moins**, ... On peut maintenant écrire

```
enum Operateur symbole = getchar();
    ...

switch (symbole)
{
    case Plus:
        ...
```

Certains compilateurs non standard émettent un message d'avertissement lors d'une conversion implicite entre un entier ou un caractère et un symbole d'énumération :

```
"oper.c", line 19: warning: enumeration type clash, operator =
```

Il est alors préférable d'effectuer une conversion explicite (voir 3.3.2) :

```
enum Operateur symbole = (enum Operateur) getchar();
```

Il n'existe pas d'opérateur spécifique défini sur les énumérations, comme *suivant* ou *précédent* par exemple. Dans le cas où les constantes sont codées par des entiers consécutifs, il est possible d'effectuer des additions et des soustractions, modulo le nombre de constantes composant l'énumération. Cela reste cependant peu pratique à manipuler. Les énumérations sont donc essentiellement destinées à la définition de constantes internes à un programme, sur lesquelles ne sont effectuées ni d'entrées-sorties, ni d'opérations autres que des affectations et des comparaisons.

Pointeurs

Un **pointeur** est une variable ou une constante dont la valeur est une **adresse**. L'adresse d'un objet est indissociable de son type. On parlera donc de *"pointeur de caractères"*, de *"pointeur d'entiers"*, etc.

L'opération fondamentale sur les pointeurs est l'**indirection**, c'est-à-dire l'évaluation de l'objet sur lequel il pointe. Le résultat de l'indirection dépend du type sur lequel est construit le pointeur.

Considérons les deux pointeurs représentés sur la figure 2.1. Le premier est un pointeur de caractères et le second un pointeur d'entiers longs (nous faisons ici l'hypothèse que la taille de ces deux types est respectivement d'un octet et de quatre octets), et on suppose que les deux pointeurs contiennent une même adresse α. Bien que contenant une même quantité numérique, ils n'ont pas la même valeur. En effet, une indirection effectuée sur l'un ou l'autre des deux pointeurs donne des résultats différents (le caractère d'adresse α dans le premier cas, et l'entier long d'adresse α dans le second).

Pour chaque type construit $\langle$*type*$\rangle$, on peut définir un nouveau type "pointeur de $\langle$*type*$\rangle$". Nous détaillons la construction des pointeurs dans la section 2.3 et leur mise en œuvre dans le chapitre 6.

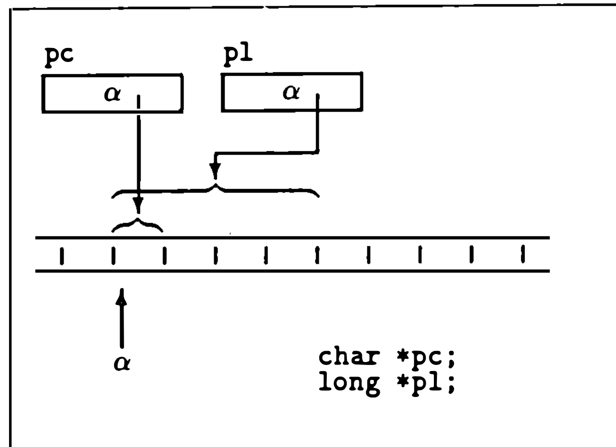


Figure 2.1: Valeur d'un pointeur

2.1.3 Types prédéfinis

En plus des types de base, l'environnement standard comporte certains types prédéfinis dans des fichiers en-tête. Il ne s'agit donc pas de constructions du langage C pur. Cependant, nous avons choisi de présenter ici trois de ces types, du fait qu'ils sont associés à des constructions du langage. Les autres types prédéfinis ne sont pas présentés dans ce chapitre. Ils sont listés dans l'annexe A.

Les trois types que nous allons voir sont définis dans le fichier en-tête `<stddef.h>`. Le premier est le type

`ptrdiff_t`

Il doit être utilisé pour stocker la différence de deux pointeurs. Ce type, généralement équivalent à `int`, peut dépendre de l'implémentation.

Le second est le type

`size_t`

Il correspond au type d'une expression construite avec l'opérateur `sizeof` (voir 3.3.6) permettant de connaître la taille d'un objet quelconque. Ce type est toujours défini comme le plus grand type entier utilisable. Il est raisonnable de l'utiliser également pour déclarer une variable codant la taille d'un vecteur.

Le dernier est le type

`wchar_t`

implémentant le concept de **caractères étendus** (en anglais *wide char*). Il s'agit de caractères dont la taille est supérieure à un octet, adaptables à des langues, comme le Japonais ou le Chinois, dont l'écriture comporte plusieurs milliers de signes. Un caractère étendu représente le codage numérique interne d'un caractère ou d'un idéogramme. Afin de permettre sa manipulation, en particulier par les fonctions standard de manipulation de chaînes de caractères, un caractère étendu est associé à un **multi-caractère** (en anglais *multibyte*), qui est sa représentation externe. Un multi-caractère est une suite de caractères standard, c'est-à-dire un vecteur de caractères, ne contenant pas le caractère `\0`. Les opérations de codage et de décodage sont effectuées par des fonctions de bibliothèque dont le comportement est propre à chaque implémentation : `mbtowc` (*multibyte to wide char*), `wctomb` (*wide char to multibyte*),

`mblen` (*multibyte length*), etc.

Si deux caractères étendus e_1 et e_2 sont tels que $e_1 < e_2$ pour l'ordre lexicographique local, c'est-à-dire celui défini sur l'ensemble de tous les caractères étendus de l'implémentation locale, alors les multi-caractères associés m_1 et m_2 vérifient $m_1 < m_2$ pour l'ordre lexicographique standard. Cela permet de faire opérer les fonctions standard de comparaison de chaînes sur des chaînes de multi-caractères.

2.2 Mécanismes de construction d'objets

Nous allons considérer dans cette section quelques exemples de déclarations. La forme générale d'une déclaration est

⟨*type-de-référence*⟩ ⟨*construction*⟩ [, ⟨*construction*⟩ ...];

Le cas le plus simple est celui où ⟨*type-de-référence*⟩ est un type de base, et ⟨*construction*⟩ un identificateur. Par exemple :

```
char caractere_lu;
int  premier, milieu, dernier;
```

définissent une variable caractère et trois variables entières. L'ensemble des constructions permettant de former des objets composés est donné sur le tableau 2.2. Les trois premiers sont les constructeurs de **pointeur**, de **vecteur** et de **fonction**. On utilise ces constructeurs de la façon suivante

⟨*type-de-référence*⟩ ⟨*vecteur*⟩[⟨*taille*⟩]
⟨*type-de-référence*⟩ ⟨*fonction*⟩ (⟨*type-paramètres*⟩)
⟨*type-de-référence*⟩ * ⟨*pointeur*⟩

Syntaxe	Objet construit
*	pointeur
[]	vecteur
()	fonction
struct	agrégat (structure)
union	union disjointe

Tableau 2.2: Constructeurs d'objets

Par exemple, les trois lignes

```
char etat[NB_ETATS];
char *caractere_courant;
char lire_caractere(void);
```

déclarent un vecteur `etat` de `NB_ETATS` caractères, un pointeur de caractères de nom `caractere_courant`, et une fonction sans paramètre `lire_caractere` retournant un caractère. Le premier élément d'un vecteur est l'élément d'indice

zéro. Les éléments du vecteur `etat` sont :

$$\begin{pmatrix} \texttt{etat[0]} \\ \texttt{etat[1]} \\ \vdots \\ \texttt{etat[NB_ETATS} - 1\texttt{]} \end{pmatrix}$$

Nous appellerons ces constructeurs des **constructeurs homogènes**. On peut les composer entre eux. Par exemple,

```
float transformation[NB_LIGNES][NB_COLONNES];
```

définit un vecteur de `NB_LIGNES` vecteurs contenant chacun `NB_COLONNES` nombres réels, c'est-à-dire une matrice `NB_LIGNES`×`NB_COLONNES` de réels, et

```
char **argv;
```

définit un pointeur de pointeurs de caractères. Lorsqu'on combine les constructeurs `[]` et `*`, on obtient un vecteur dont chaque élément est un pointeur. De même, la combinaison de `()` et `*` donne une fonction dont la valeur de retour est un pointeur. Les déclarations suivantes

```
char *argv[];
char *lire_chaine(void);
```

définissent respectivement un vecteur `argv` de pointeurs de caractères, et une fonction `lire_chaine` retournant un pointeur de caractères. On peut utiliser des parenthèses pour modifier la priorité de ces constructeurs; par exemple, les deux déclarations suivantes

```
int (* adresse_de_la_liste)[TAILLE_LISTE];
void (* recuperation_erreur)(void);
```

définissent respectivement le pointeur `adresse_de_la_liste` sur un vecteur de `TAILLE_LISTE` entiers, et le pointeur `recuperation_erreur` sur une fonction de type `void`.

Il existe plusieurs manières de construire de nouveaux types de référence. Une première façon consiste à nommer une construction au moyen du constructeur `typedef`. Nous détaillons ce mécanisme en 2.5.1. Une autre façon consiste à effectuer des regroupements d'objets en **structures**. Par exemple, la structure `struct mot` suivante

```
struct mot
{
    int longueur;
    char chaine[TAILLE_MAX_MOT];
};
```

est un regroupement de deux objets :

- un entier,
- un vecteur de `TAILLE_MAX_MOT` caractères.

Ces deux objets sont appelés les **champs** de la structure. Le premier champ est identifié par le nom `longueur` et le second par le nom `chaine`. Le litté-

ral struct mot constitue un nouveau type de référence utilisable dans une déclaration. Par exemple, la suite de déclarations

```
struct mot un_mot;
struct mot liste_mots[NB_MOTS];
```

définit une structure struct mot de nom un_mot, et un vecteur liste_mots de NB_MOTS structures. L'expression élémentaire un_mot.longueur a pour valeur le contenu du champ longueur de la structure un_mot, et l'expression liste_mots[0].chaine[0] vaut le premier caractère du champ chaine du premier élément du vecteur de structures liste_mots.

2.3 Constructeurs homogènes

Ces constructions regroupent les définitions de vecteurs, de pointeurs et de fonctions. Considérons tout d'abord le cas de la déclaration d'un vecteur de pointeurs de caractères. Une syntaxe naturelle pourrait être

```
vpc : array[TVPC] of pointer of char;
```

Nous visualiserons la structure syntaxique de cette déclaration au moyen du schéma suivant :

```
(        [ ]  )
(         |   )
( vpc,    *   )
(         |   )
(       char  )
```

qu'on peut encore écrire (vpc, [ ](* (char))) Ces deux schémas ont la même signification : vpc est de type *"vecteur de pointeurs de caractères"* (pour simplifier l'écriture, nous n'avons pas fait figurer la dimension du vecteur dans le schéma).

La notation que nous avons qualifiée plus haut de *naturelle* consiste à calquer la syntaxe de la déclaration sur le schéma syntaxique précédent. Elle consiste à construire un type composé par applications successives de constructeurs sur un type de référence, construction qu'on préfixe ensuite par le nom de l'objet.

En C, on applique au contraire des opérateurs de construction d'objet sur un nom d'objet, c'est-à-dire un identificateur, et on préfixe ensuite la construction avec un type de référence. Ainsi, un vecteur de caractères s'écrit char v[TV] et non v: char[TV] ou dans une syntaxe plus verbeuse

```
variable v is array[TV] of char
```

En fait, cette écriture correspond à un *renversement* du schéma syntaxique. Ainsi, si on reconsidère l'exemple du vecteur de pointeurs de caractères, on applique tout d'abord le constructeur [] sur l'identificateur vpc :

```
vpc[TVPC]
```

puis le constructeur * sur le résultat :

```
*(vpc[TVPC])
```

Enfin, on préfixe cette construction par le type de référence qui dans ce cas est char, ce qui donne finalement

```
char   *(vpc[TVPC])
```

Le constructeur `[]` est prioritaire par rapport au constructeur `*`. Dans la déclaration précédente, les parenthèses sont par conséquent inutiles, et on peut directement écrire

```
char   *vpc[TVPC];
```

On peut définir de façon informelle la suite de réécritures permettant de passer de l'arbre de syntaxe à la syntaxe de la déclaration. Sur l'exemple précédent, cette suite de réécritures est :

$$\left(\texttt{vpc},\ \begin{array}{c}[\,]\\ |\\ *\\ |\\ \texttt{char}\end{array}\right) \longrightarrow \left(\texttt{vpc[TVPC]},\ \begin{array}{c}*\\ |\\ \texttt{char}\end{array}\right)$$
$$\longrightarrow (*\texttt{vpc[TVPC]},\ \texttt{char}\ )$$
$$\longrightarrow (\texttt{char}\ *\texttt{vpc[TVPC]},\ \varepsilon\ )$$

(ε désigne le mot vide, c'est-à-dire la suite de caractères de longueur nulle.)

Remarque 7 *On peut vérifier la validité d'une construction en interprétant la construction comme une expression, en évaluant son type, et en vérifiant qu'on retrouve le type de référence.*

On vérifie que l'évaluation du type de `*vpc[TVPC]` donne le type `char`. En effet, si `vpc` est un vecteur de pointeurs de caractères, alors `vpc[TVPC]` est un pointeur de caractères, et `*vpc[TVPC]` est un caractère.

Considérons maintenant un second exemple dans lequel on définit un objet `pvi` comme un pointeur vers un vecteur de `NB_INT` entiers courts. Le schéma décrivant cette construction est :

$$\left(\texttt{pvi},\ \begin{array}{c}*\\ |\\ [\,]\\ |\\ \texttt{short}\end{array}\right)$$

Comme dans le cas précédent, on passe de l'arbre de syntaxe à la déclaration par trois réécritures successives :

$$\left(\texttt{pvi},\ \begin{array}{c}*\\ |\\ [\,]\\ |\\ \texttt{short}\end{array}\right) \longrightarrow \left(*\texttt{pvi},\ \begin{array}{c}[\,]\\ |\\ \texttt{short}\end{array}\right)$$
$$\longrightarrow ((*\texttt{pvi})\texttt{[NB_INT]},\ \texttt{short}\ )$$
$$\longrightarrow (\texttt{short}\ (*\texttt{pvi})\texttt{[NB_INT]},\ \varepsilon\ )$$

La syntaxe de cette déclaration est donc

```
short (*pvi)[NB_INT];
```

On doit, dans ce cas, conserver les parenthèses pour forcer les règles de priorité, lors de l'application du constructeur `vecteur` sur la construction `*pvi`.

Tout ce que nous venons de dire sur le constructeur `[]` s'applique également au constructeur `()`, qui a le même niveau de priorité. Ainsi, la déclaration d'une fonction `f` retournant un pointeur d'entiers, dont le schéma associé est :

```
/     ( )  \
|      |   |
| f,   *   |
|      |   |
\     int  /
```

s'écrit `int *f();` alors que celle d'un pointeur `pf` vers une fonction entière, représentée par le schéma

```
/       *   \
|       |   |
| pf,  ( )  |
|       |   |
\      int  /
```

s'écrit `int (*f)();`

Nous avons mentionné que les constructeurs de vecteurs et de fonctions ont même priorité. On pourrait considérer que

```
float objet_impossible[16]()
```

construit un vecteur de 16 fonctions réelles, et que

```
float autre_objet_impossible()[16]
```

déclare une fonction retournant un vecteur de 16 nombres réels. Si ces deux constructions sont syntaxiquement justes, elles sont par contre sémantiquement incorrectes. Il n'est pas possible de construire de vecteur de fonctions[1] et une fonction ne peut retourner qu'un objet élémentaire ou une structure. La construction

```
void encore_un_objet_impossible()()
```

est, par conséquent, elle aussi interdite.

Nous allons considérer pour terminer une construction plus complexe : un *vecteur* `tbf` *de pointeurs de fonctions retournant un pointeur d'entiers.* Le premier constructeur à appliquer est le constructeur *vecteur* :

```
tbf[dim]
```

On applique ensuite le constructeur *pointeur*; ce dernier étant d'un niveau de priorité plus faible que *vecteur*, il n'est pas nécessaire d'utiliser de parenthèses :

```
*tbf[dim]
```

Le troisième constructeur appliqué est le constructeur *fonction* :

```
(*tbf[dim])()
```

[1] Il est par contre possible de construire des vecteurs de pointeurs de fonctions.

Cette fois, il est nécessaire d'utiliser des parenthèses pour conserver l'information que *pointeur* doit s'appliquer avant *fonction*. Enfin, on applique une nouvelle fois le constructeur *pointeur*, à nouveau sans parenthèses, *fonction* étant prioritaire, ce qui donne :

```
*(*tbf[dim])();
```

La déclaration complète s'obtient en préfixant cet objet par le type de base, `int` dans ce cas, et on obtient l'instruction :

```
int  *(*tbf[dim])();
```

2.4 Constructeurs hétérogènes

2.4.1 Regroupement d'objets : les structures

La construction hétérogène la plus naturelle est le regroupement séquentiel d'objets pour former un enregistrement appelé, en C, une **structure**. Un exemple très classique est la définition d'un nombre complexe $a + bi$; en C il s'écrit :

```
struct complexe
{
    double a;
    double b;
};
```

La syntaxe générale de cette construction est :

```
struct [⟨nom⟩]
{
    ⟨liste-de-déclarations⟩
}
```

Il existe plusieurs façons de l'utiliser. La déclaration suivante :

```
struct
{
    short x;
    short y;
    long  couleur;
} pixel;
```

déclare un objet, de nom `pixel`, et composé de trois champs : deux entiers courts et un entier long. On accède aux champs de la structure au moyen de l'opérateur `.` :

```
pixel.x
pixel.y
pixel.couleur
```

On peut donner un nom à la structure ainsi définie, ce qui facilite sa réutilisation dans une autre déclaration. Par exemple, la déclaration :

```
struct pixel
{
    short x;
    short y;
    long  couleur;
};
```

associe le nom `struct pixel` à la construction précédente. La forme `struct pixel` est maintenant utilisable comme un type de référence :

```
struct pixel nuage[TAILLE_MAX_NUAGE];
malloc(sizeof(struct pixel));
```

Il est possible de combiner la définition d'une nouvelle structure qui est une définition de type, avec des définitions d'objets. Dans l'exemple suivant, on effectue simultanément la définition du type `struct pixel` et la définition de l'objets `nuage` :

```
struct pixel
{
    short x;
    short y;
    long  couleur;
} nuage[TAILLE_MAX_NUAGE];
```

Cependant, nous éviterons cette dernière forme et nous retiendrons la règle suivante :

☹ *Il est préférable de séparer, lors d'une déclaration de structure, les définitions de type des définitions d'objet.*

2.4.2 Champs de bits

Il est possible de découper un entier en **champs de bits** dont la longueur peut varier de 1 bit à un nombre dépendant de l'implémentation. Ces champs sont spécifiés en utilisant le constructeur de structures.

La syntaxe que nous retiendrons pour la spécification de champs de bits est la suivante :

```
struct [⟨nom⟩]
{
    unsigned [⟨ident⟩]: ⟨longueur⟩;
    ...
}
```

Par exemple, la déclaration

```
struct cellule
{
    unsigned actif   :  1;
    unsigned type    :  3;
    unsigned valeur  : 14;
    unsigned suivant : 14;
}
```

définit le découpage d'un entier de 32 bits en quatre champs. Si `c` est une variable déclarée par :

```
struct cellule c;
```

le champ `c.actif` peut prendre les valeurs 0 ou 1, le champ `c.type` les valeurs comprises entre 0 et 7, et les deux derniers champs, des valeurs comprises entre 0 et 16383. L'exécution de la suite d'instruction

```
c.type = 7;
printf("%d + 1 = ", c.type);
c.type++;
printf("%d\n", c.type);
```

produit l'affichage :

```
7 + 1 = 0
```

Ces constructions ne sont pas portables. En particulier, la taille maximum d'un champ dépend de la machine; c'est en général celle d'un entier long.

2.4.3 Unions d'objets

Le constructeur `union` permet de représenter par un même type générique plusieurs types différents. Sa syntaxe générale est la même que celle du constructeur `struct` :

```
union [⟨nom⟩]
{
    ⟨liste-de-déclarations⟩
}
```

L'exemple suivant définit un type générique pouvant être soit un entier, soit un réel, soit un nombre complexe :

```
union valeur
{
    long entier;
    double reel;
    struct complexe complexe;
};
```

Un objet `v` déclaré de ce type :

```
union valeur v;
```

peut être utilisé comme un entier :

```
v.entier
```

comme un réel :

```
v.reel
```

ou comme une structure représentant un nombre complexe :

```
v.complexe
```

Dans ce dernier cas, on peut accéder indifféremment à sa partie réelle ou à sa partie imaginaire :

```
sqrt(v.complexe.a * v.complexe.a + v.complexe.b * v.complexe.b)
```

La figure 2.2 représente l'organisation de la variable `v`. Dans cette implémentation, elle occupe seize octets : c'est la taille du plus grand de ses champs. Les quatre premiers sont utilisés pour coder le champs `v.entier`, les huit premiers le champ `v.reel`, et enfin les seize octets sont utilisés pour le codage du champ `v.complexe`.

La taille d'une structure est au minimum la somme des tailles de ses champs (elle peut être supérieure à cause de réalignements effectués par le compilateur – voir 3.3.5); celle d'une union est égale à la plus grande des tailles de ses champs.

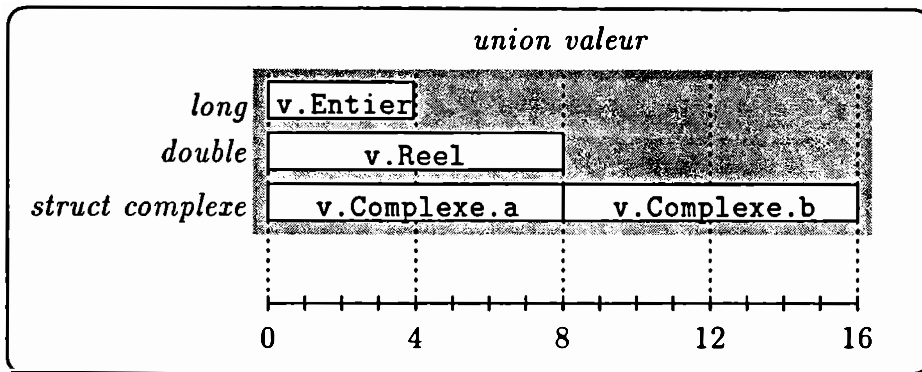


Figure 2.2: Implémentation du type `union valeur`

La même construction peut coder plusieurs types différents, mais une donnée ne peut changer de type. Si une variable de type `union valeur` a été initialisée sous la forme d'un entier, son utilisation comme un réel ou comme un complexe n'a pas de sens. On associe généralement à un type union une étiquette indiquant, pour chaque variable de ce type, celle des différentes interprétations qu'il convient d'utiliser. Sur l'exemple, on définit trois constantes, `Entier`, `Reel` et `Complexe` permettant l'étiquetage du type d'une variable `union valeur`, puis on regroupe la variable et son étiquette dans une structure.

```
enum type_numerique {Entier, Reel, Complexe};

struct nombre
{
    enum  type_numerique type;
    union valeur valeur;
};
```

Voici, pour finir, l'initialisation d'un nombre complexe `n` avec le nombre imaginaire i.

```
struct nombre n;

n.type = Complexe;
n.valeur.complexe.a = 0.0;
n.valeur.complexe.b = 1.0;
```

Si on utilise simultanément plusieurs interprétations différentes sur un même objet de type union, le résultat n'est pas portable. Considérons l'exemple suivant :

```
union attention
{
    char c[2];
    short s;
};
union attention danger;
```

Si on utilise cette construction pour accéder à la partie haute ou à la partie basse d'un entier court, comme dans la suite d'instructions

```
danger.s = n;
partie_basse = danger.c[0];
partie_haute = danger.c[1];
```

le résultat dépend de la machine sur laquelle s'exécute le programme[2]. Une telle construction peut, de ce fait, servir à tester le type de machine sur lequel tourne le programme.

2.5 Définition de types

Nous avons vu que les constructeurs `struct` et `union` permettent de construire et de nommer de nouveaux types. On peut également donner un nom à un type au moyen du constructeur `typedef`. On peut enfin construire de nouveaux types au moyen d'expressions de type, à partir d'un type de référence et des constructeurs homogènes.

2.5.1 Le constructeur typedef

La syntaxe du constructeur `typedef` est

`typedef` ⟨*declaration*⟩

Si l'instruction

⟨*type*⟩ ⟨*construction*⟩

définit l'objet de nom ⟨*nom*⟩ et de type ⟨*type*⟩, alors

`typedef` ⟨*type*⟩ ⟨*construction*⟩

donne le nom ⟨*nom*⟩ au type ⟨*type*⟩. Par exemple

```
char couleur[3];
```

définit un vecteur de trois caractères de nom `couleur`, et

```
typedef char couleur[3];
```

donne le nom `couleur` au type *vecteur de trois caractères*. Ce nouveau nom constitue un nouveau type de référence. Par exemple, la suite de déclarations

```
couleur bleu;
couleur table_de_couleurs[TAILLE_TABLE_DE_COULEURS];
```

[2]Selon les machines, les mots sont codés avec l'octet de poids fort avant ou après l'octet de poids faible.

définit un vecteur de trois caractères de nom `bleu` et un vecteur, de nom `table_de_couleurs`, et composé de `TAILLE_TABLE_DE_COULEURS` vecteurs de trois caractères.

Voici un autre exemple d'utilisation du constructeur `typedef` avec des définitions de structures. Considérons la définition :

```
struct cellule
{
    struct cellule *suivant;
    struct cellule *precedent;
    char   *contenu;
};
```

Elle définit le type de nom `struct cellule`, et la déclaration

```
struct cellule c;
```

définit un objet de ce type; par conséquent la déclaration

```
typedef struct cellule type_cellule;
```

définit le mot `type_cellule` comme étant un synonyme de `struct cellule`. Il est également possible d'écrire

```
typedef struct cellule cellule;
```

ce qui rend `cellule` et `struct cellule` synonymes, ce qui permet d'écrire simplement

```
cellule c;
cellule *p;
```

à la place de

```
struct cellule c;
struct cellule *p;
```

On pourra effectuer cette construction systématiquement[3], lors d'une déclaration de structure :

```
struct ⟨Nom⟩
{
    ...
};
typedef struct ⟨Nom⟩ ⟨Nom⟩;
```

Il faut noter que la construction `typedef` ne s'applique que sur un type déjà défini. Par conséquent, on ne peut écrire

```
typedef struct
{
    cellule *suivant;
    cellule *precedent;
    char    *contenu;
} cellule;
```

[3]Cette construction est parfois effectuée automatiquement en langage C++; par conséquent, bien que le langage C soit en théorie contenu dans le langage C++, la compilation de ces constructions par un compilateur C++ peut provoquer des messages d'erreur.

c'est-à-dire utiliser le nom de type `cellule` dans la définition de la structure. De toutes façons, cette construction est en contradiction avec le principe énoncé page 59.

2.5.2 Expressions de type

On peut exprimer un type par une construction analogue à la déclaration d'un objet. On a vu en 2.2 qu'on construit un objet en appliquant les opérateurs homogènes *, [] et () sur le nom de l'objet, et en préfixant le tout par un type de référence.

On obtient l'expression du type d'un objet en effaçant le nom de cet objet dans la construction qui le définit. Par exemple, le type de l'objet

`vecteur_de_chaines`

défini par

`char *vecteur_de_chaines[NBCHN];`

est `char *[NBCHN]`. En d'autres termes, on obtient l'expression du type d'une construction en appliquant cette construction sur l'identificateur ε, c'est-à-dire le mot vide. Par exemple, le type *pointeur vers un vecteur de 10 caractères* s'obtient par la construction `char(*ε)[10]`, et donc s'écrit

`char(*)[10]`

Voici quelques autres exemples d'expressions de types :

- `char(*)(void)` : pointeur de fonctions sans paramètre retournant un caractère;
- `char(*[256])(char *)` : vecteur de 256 pointeurs de fonctions recevant un pointeur de caractères en paramètre et retournant un caractère;
- `int *(*[])(int, int)` : vecteur de pointeurs de fonctions recevant deux entiers en paramètres et retournant un pointeur d'entiers.

Le programme suivant fait imprimer la taille de plusieurs types définis par des expressions de type. Il utilise l'opérateur d'expression `sizeof` qui calcule la taille d'un type (voir 3.3.6).

```
========== types.c ==========
#include <stdio.h>

main()
{
    printf("\tTailles de\n");
    printf("\t----------\n");
    printf("char *       : %d octets\n", sizeof(char *));
    printf("char *[10]   : %d octets\n", sizeof(char *[10]));
    printf("char(*)[10]  : %d octets\n", sizeof(char(*)[10]));
}
========== types.c ==========
```

Dans cette implémentation, la taille d'un pointeur est 4 octets. Par conséquent, la taille d'un vecteur de 10 pointeurs est 40 octets. La troisième expression définit un pointeur vers un vecteur de 10 caractères. La taille d'un objet de ce type est celle d'un pointeur.

```
$ types
       Tailles de
       ----------
char *       : 4 octets
char *[10]   : 40 octets
char(*)[10]  : 4 octets
$
```

2.6 Implémentation et visibilité

La durée de vie d'une variable dépend de son implémentation : permanente ou temporaire. L'implémentation d'une variable, tout comme sa visibilité, est liée à son niveau de déclaration. Implémentation et visibilité implicites peuvent être modifiées, au moyen des mots-clés

```
auto      register
const     static
extern    volatile
```

2.6.1 Niveau d'une variable

Le **niveau** d'une variable est déterminé par l'emplacement de sa déclaration dans le programme. Une variable peut être

- une **variable de niveau zéro** lorsqu'elle est déclarée à l'extérieur de toute fonction;
- une **variable de bloc**, ou variable de **niveau** n ($n \geq 1$) lorsqu'elle est déclaré au début d'un bloc.

Une troisième famille de variables est celle des **paramètres de fonction**, qu'on peut classer entre les variables de niveau zéro et un. Considérons le fichier suivant :

```
caractere_suivant.c
#include "caractere_suivant.h"

int caractere_a_lire = -1;

int caractere_suivant()
{
    if (caractere_a_lire != -1)
    {
        int c = caractere_a_lire;

        caractere_a_lire = -1;
        return c;
    }
    return getchar();
}

void relire_caractere(char c)
```

```
{
    caractere_a_lire = c;
}
```
caractere_suivant.c

La variable caractère `caractere_a_lire` est une variable de niveau zéro. Elle est déclarée à l'extérieur de toute fonction. La fonction `relire_caractere` possède un paramètre qui est le caractère `c`. Enfin, la variable `c` déclarée dans la fonction `caractere_suivant` est une variable de bloc. Réécrivons cette fonction en ne conservant que les déclarations :

```
int caractere_suivant() { { int c; } }
```

La variable `c` est déclarée à l'intérieur de deux blocs imbriqués : elle est de niveau 2. Plus généralement, le niveau d'une variable de bloc est égal à la profondeur du bloc dans lequel elle est déclarée.

2.6.2 Implémentation des variables

Durée de vie

Il existe deux implémentations possibles pour une variable :

- **permanente** (ou *statique*) : l'emplacement mémoire utilisé par la variable est alloué lors de la compilation du programme, et demeure inchangé durant son exécution;
- **temporaire** (ou *dynamique*) : l'emplacement mémoire de la variable est alloué lors de l'appel de la fonction dans laquelle elle est définie, et est libéré lors du retour de cette fonction.

De plus, les variables permanentes sont initialisées à zéro par le compilateur.

Une variable de niveau zéro est toujours permanente. Un paramètre de fonction est toujours une variable temporaire; il est créé au moment de l'appel de la fonction, et disparaît lors du retour. Par défaut, une variable de bloc est temporaire; elle n'a d'existence réelle que durant l'exécution des instructions du bloc dans lequel elle est définie. Si on sort d'un bloc contenant des définitions de variables temporaires, puis qu'on y entre à nouveau durant l'exécution du programme, ces variables peuvent avoir n'importe quelle valeur.

L'espace mémoire est alloué globalement pour toutes les variables temporaires d'une fonction lors de l'entrée dans cette fonction et est libéré lors du retour. L'emplacement d'une variable temporaire peut éventuellement être partagé avec d'autres variables temporaires appartenant à des blocs différents de la même fonction.

Il est possible de rendre permanente une variable de bloc au moyen du mot-clé `static`. Elle se comporte alors comme une variable de niveau 0 : son emplacement n'est pas libéré lors du retour de la fonction dans laquelle elle est définie, et elle conserve sa valeur entre deux appels. Exemple :

permanente.c

```
#include <stdio.h>

int suivant(void);
```

```
main()
{
    int i;

    do
    {
        i = suivant();
        printf(" suivant() vaut: %d\n", i);
    }
    while (i < 5);
}

int suivant()
{
    static int s = 0;

    return s++;
}
```
permanente.c

On peut voir sur cet exemple que l'initialisation d'une variable permanente, ici l'initialisation à zéro de la variable `s`, s'effectue une seule fois. Chaque exécution de la fonction `suivant` retourne une nouvelle valeur.

```
$ permanente
 suivant() vaut: 0
 suivant() vaut: 1
 suivant() vaut: 2
 suivant() vaut: 3
 suivant() vaut: 4
 suivant() vaut: 5
$
```

Variables automatiques et variables registre

Il existe, en C, deux implémentations possibles pour une variable temporaire:

- dans la pile d'exécution du programme : ce sont les variables **automatiques**;
- dans un registre de donnée de la machine.

Par défaut, un variable temporaire est automatique. On peut également le spécifier explicitement au moyen du mot-clé `auto`. Il n'est utilisable qu'avec des variables temporaire. Une variable de niveau zéro étant toujours permanente, on ne peut écrire

```
auto   int i;
```

au niveau zéro. Cette erreur est signalée par le compilateur `gcc` par le message :

```
automatic.c:1: top-level declaration of 'i' specifies 'auto'
```

ou par le compilateur traditionnel par :

```
"automatic.c", line 1: illegal class for i
```

Lorsqu'une variable est très utilisée, il peut être avantageux de demander au compilateur qu'elle soit, dans la mesure du possible, rangée dans un registre de la machine. Cette requête, qui ne peut s'appliquer qu'à des variables temporaires, sera satisfaite s'il reste des registres disponibles au format de cette donnée. Elle est effectuée au moyen du mot-clé `register` préfixant la déclaration. Exemple :

```
register int  ind1, ind2;
register char *p;
```

A chaque nouvel appel de fonction, les registres utilisés dans la fonction appelante sont sauvegardés dans la pile d'exécution du processus, et restaurés au retour de l'appel. Le nombre maximum de registres disponibles est dépendant de la machine et du compilateur. Ce nombre étant limité, une spécification `register` n'est pas nécessairement satisfaite; il faut sélectionner dans chaque fonction les variables les plus utilisées, comme par exemple les indices de boucle.

Le programme suivant effectue un calcul coûteux utilisant plusieurs variables de boucle.

sans_reg.c

```
#include <stdio.h>

main(int argc, char **argv)
{
    int i;
    int s = 0;
    int n = argc == 2 ? atoi(argv[1]) : 0;

    for (i=1; i<=n; i++)
    {
        int j;

        for (j=0; j<n; j+=i)
            if ((i+j) % 2)
                s++;
    }
    printf(" %d\n", s);
}
```

sans_reg.c

Cette seconde version réalise le même calcul. La seule différence réside dans la déclaration des variables avec la spécification `register`.

avec_reg.c

```
main(int argc, char **argv)
{
    register int i;
    register int s = 0;
    register int n = argc == 2 ? atoi(argv[1]) : 0;
```

```
    for (i=1; i<=n; i++)
    {
        register int j;

        for (j=0; j<n; j+=i)
            if ((i+j) % 2)
                s++;
    }
    printf(" %d\n", s);
}
```

avec_reg.c

On peut mesurer le temps de chaque exécution au moyen de la commande TIME(1). Sur une station *Sparc ELC* sous système *SUN-OS*, la version implémentée avec des registres est plus deux fois plus rapide en temps UC.

```
$ time sans_reg 100000
 341683
        2.9 real          1.5 user          0.0 sys
$ time avec_reg 100000
 341683
        1.8 real          0.7 user          0.0 sys
$
$ time sans_reg 1000000
 3992484
       18.7 real         17.6 user          0.0 sys
$ time avec_reg 1000000
 3992484
       10.3 real          8.1 user          0.1 sys
$
```

Le *temps UC* consommé par le processus est celui étiqueté `user`, le second affiché par `time`. Le premier, étiqueté `real`, indique le temps réellement écoulé entre le lancement de la commande et sa terminaison

2.6.3 Optimisation du code généré

L'optimisation de l'utilisation de variables au moyen de registres ne devrait pas être à la charge du programmeur. Les compilateurs C disposent en général d'un optimiseur de code qu'on peut invoquer au moyen de l'option `-O`. Compilés avec cette option, les temps d'exécution des deux programmes précédents sont très voisins. On notera par contre l'amélioration, d'environ un facteur 1.7, entre la version non optimisée avec registres et les versions optimisées.

```
$ gcc -g -O avec_reg.c -o avec_reg
$ gcc -g -O sans_reg.c -o sans_reg
$
$ time sans_reg 100000
 341683
        1.6 real          0.4 user          0.0 sys
$ time avec_reg 100000
 341683
        1.0 real          0.4 user          0.0 sys
```

```
$
$ time sans_reg 1000000
 3992484
        6.0 real        4.7 user        0.1 sys
$ time avec_reg 1000000
 3992484
        5.6 real        4.7 user        0.0 sys
$
```

Remarque 8 *Le compilateur* gcc *possède la particularité de supporter simultanément les options* -g *de déboguage symbolique et* -O *d'optimisation de code. Ce n'est généralement pas le cas des autres compilateurs :*

```
$ cc -g -O avec_reg.c -o avec_reg-O
cc: Warning: -O conflicts with -g. -O turned off.
$
```

L'attribution de registres de données pour les variables temporaires n'est pas le seul travail effectué par l'optimiseur de code. Il recherche également les instructions inutiles ou redondantes, comme par exemple une affectation répétée dans une boucle et pouvant être effectuée une seule fois à l'extérieur. L'optimiseur peut ainsi déplacer ou supprimer certaines instructions. Considérons par exemple le programme suivant :

```
========== optimisation.c ==========
main()
{
    int var = -1;
    int i = 1;

    while (i)
        i = var;
}
========== optimisation.c ==========
```

L'option -S de gcc permet d'effectuer seulement la compilation du fichier source. Le fichier produit est la traduction du code C en assembleur (ici l'assembleur 68000); il est suffixé par .s. Nous l'avons listé au moyen de la commande CAT(1) qui liste le contenu d'un fichier. L'option -n numérote les lignes.

```
$ gcc -S optimisation.c; cat -n optimisation.s
     1  #NO_APP
     2  gcc_compiled.:
     3  .text
     4          .even
     5  .globl _main
     6  _main:
     7          link  a6,#-8
     8          moveq #-1,d0
     9          movel d0,a6@(-4)
```

```
     10          moveq #1,d0
     11          movel d0,a6@(-8)
     12  L2:
     13          tstl  a6@(-8)
     14          jeq   L3
     15          movel a6@(-4),a6@(-8)
     16          jra   L2
     17  L3:
     18  L1:
     19          unlk  a6
     20          rts
$
```

Sans trop examiner dans le détail ce programme assembleur, on peut reconnaître dans les lignes 8 et 9 la traduction de l'expression `var = -1`. En effet la variable `var` est désignée par l'expression `a6@(-4)` qui référence une adresse dans la pile d'exécution, et l'identificateur `d0` désigne un registre de données, utilisé comme registre intermédiaire pour traiter l'affectation de la valeur -1. Plus précisément, ces deux lignes effectuent le calcul[4] :

COPIER *la constante -1* DANS *le registre* d0
COPIER *le registre* d0 DANS *la variable* var

De même, les lignes 10 et 11 traduisent l'expression `i = 1`. Les lignes 12 à 17 correspondent à :

```
debut:
  if (i == 0)
      goto fin;
  i = var;
  goto debut;
fin:
```

Considérons maintenant le même programme compilé avec l'option `-O` :

```
$ gcc -S -O optimisation.c; cat -n optimisation.s
     1  #NO_APP
     2  gcc_compiled.:
     3  .text
     4          .even
     5  .globl _main
     6  _main:
     7          link  a6,#0
     8          moveq #-1,d0
     9  L4:
    10          tstl  d0
    11          jne   L4
    12          unlk  a6
    13          rts
$
```

On remarque que le compilateur a attribué le registre de données `d0` à la

[4]Les mnémoniques `moveq` et `movel` signifient respectivement *move quick* et *move long data*.

variable `i`. L'instruction `i = var;` a été supprimée; elle a été jugée inutile, du fait que `var` n'est jamais modifiée. La variable `i` est directement initialisée avec `-1`. La boucle est réduite aux lignes 9 à 11. Les lignes 8 à 11 correspondent à :

```
    i = -1;
debut:
    if (i != 0)
        goto debut;
```

2.6.4 Variables volatiles

Il existe un cas pour lequel l'optimisation du code par l'optimiseur peut conduire à des erreurs. Il s'agit de variables susceptibles d'être modifiées indépendamment du déroulement normal du programme : variable modifiée sur réception d'une interruption ou d'un signal, variable implantée à une adresse directement utilisée par le *hardware* de la machine, etc. Pour signaler au compilateur qu'une variable peut changer de valeur même si cela n'apparaît pas explicitement dans le source du programme, on utilise le mot-clé `volatile`.

Considérons le programme de la section précédente, dans lequel la variable `var` est déclarée volatile :

```
======== volatile.c ========
main()
{
    volatile int var = -1;
    int i = 1;

    while (i)
        i = var;
}
======== volatile.c ========
```

Le code assembleur généré après optimisation est le suivant :

```
$ gcc -S -O optimisation.c; cat optimisation.s
     1  #NO_APP
     2  gcc_compiled.:
     3  .text
     4          .even
     5  .globl _main
     6  _main:
     7          link  a6,#-4
     8          moveq #-1,d1
     9          movel d1,a6@(-4)
    10  L4:
    11          movel a6@(-4),d0
    12          jne   L4
    13          unlk  a6
    14          rts
$
```

La variable `var` est rangée dans la pile d'exécution; elle est codée par l'expression `a6@(-4)`. La variable `i` est rangée dans le registre de donnée `d0`. Les

lignes 8 et 9 traduisent l'expression `var = -1`; elles utilisent comme registre intermédiaire le registre de données `d1`. La boucle est gérée par les instructions 10 à 12 correspondant à :

```
debut:
  i = var;
  si non nul goto debut;
```

Le test *si non nul* porte sur la valeur affectée lors de l'instruction précédente. On voit ainsi que, contrairement au cas précédent, l'optimiseur de code a conservé l'affectation `i = var` dans la boucle, bien que la variable `var` ne soit pas explicitement modifiée.

2.6.5 Règles de visibilité

Visibilité des fonctions

Par défaut, une fonction est **globale**, c'est-à-dire appelable depuis n'importe quelle partie du programme. Si le programme est composé de plusieurs fichiers, une fonction globale peut être appelée depuis n'importe quel fichier.

Une fonction peut être rendue **locale** au moyen du mot-clé `static`. Dans ce cas, la fonction est locale au fichier dans lequel elle est définie. L'exemple qui suit illustre ce mécanisme : les deux fichiers `tri3car.c` et `tri3int.c` contiennent tous deux une fonction locale appelée `echange`. La fonction du premier module réalise l'échange de trois caractères.

```
tri3car.c
/* tri3car.c : tri de trois caracteres */

static void echange(char *, char *);

void tri3car(char *c1, char *c2, char *c3)
{
    if (*c1 > *c2)
        echange(c1, c2);
    if (*c2 > *c3)
        echange(c2, c3);
    if (*c1 > *c2)
        echange(c1, c2);
}

static void echange(char *c1, char *c2)
{
    char c;

    c = *c1, *c1 = *c2, *c2 = c;
}
tri3car.c
```

Les paramètres sont transmis par référence (voir 1.7.3 et 6.4.2). Le module suivant ne diffère avec celui-ci que dans la déclaration des variables : `int` au lieu de `char`.

```
═══════════ tri3int.c ═══════════════════════════════════
/* tri3int.c : tri de trois entiers */

static void echange(int *, int *);

void tri3int(int *c1, int *c2, int *c3)
{
    if (*c1 > *c2)
        echange(c1, c2);
    if (*c2 > *c3)
        echange(c2, c3);
    if (*c1 > *c2)
        echange(c1, c2);
}

static void echange(int *c1, int *c2)
{
    int c;

    c = *c1, *c1 = *c2, *c2 = c;
}
═════════════════════════════════════ tri3int.c ═════════
```

Le fichier `tri3.c` permet de tester ces deux fonctions.

```
═══════════ tri3.c ══════════════════════════════════════
/*   tri3.c :    test des fonctions tri3car() et tri3int() */

#include <stdio.h>

extern void tri3car(char*, char *, char *);
extern void tri3int(int*, int *, int *);

main()
{
    int i1 = 3, i2 = 2, i3 = 1;
    char c1 = 'x', c2 = 'z', c3 = 'y';

    tri3int(&i1, &i2, &i3);
    tri3car(&c1, &c2, &c3);
    printf("  - les entiers : %d, %d, %d\n", i1, i2, i3);
    printf("  - les caracteres : %c, %c, %c\n", c1, c2, c3);
}
═════════════════════════════════════════ tri3.c ═════════
```

La compilation de ces trois fichiers et l'exécution du programme résultant se déroule sans erreur :

```
$ gcc -g tri3*.c -o tri3
$
$ tri3
- les entiers : 1, 2, 3
- les caracteres : x, y, z
$
```

Si par contre on enlève les mots clés `static` qui préfixent les déclarations des

fonctions **echange**, on obtient le message d'erreur :

```
$ gcc -g tri3*.c -o tri3
_echange: ld: tri3int.o: multiply defined
$
```

La compilation séparée de chacun des trois fichiers s'est déroulée sans problème. Par contre, l'éditeur de liens détecte une double définition du symbole **echange**.

Visibilité des variables

Une variable de niveau zéro est visible dans toute la partie du fichier située après sa déclaration. Les paramètres d'une fonction sont visibles seulement dans cette fonction. Les variables définies dans un bloc sont visibles seulement dans ce bloc et les sous-blocs qu'il contient. Une déclaration de niveau n masque toute déclaration de niveau inférieur qui l'englobe, y compris les paramètres et les variables de niveau zéro. Le programme suivant met en évidence ces quelques règles de visibilité.

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

char *chaine = "chaine definie au niveau zero";
void fonction(char *);

main()
{
    printf("\nDifferentes valeurs de la variable chaine:\n");
    printf("    - dans  main : \"%s\"\n", chaine);
    fonction("chaine passee en parametre");
}

void fonction(char *chaine)
{
    printf("    - a l'entree de fonction : \"%s\"\n", chaine);
    {
        char *chaine = "chaine definie dans le bloc 1";

        printf("    - dans le bloc 1 : \"%s\"\n", chaine);
    }
    {
        int chaine = 12345;

        printf("\nRemarque: dans le bloc 2, chaine est un");
        printf(" entier valant %d\n\n", chaine);
    }
}
visibilite.c
```

```
$ visibilite

Differentes valeurs de la variable chaine:
    - dans  main : "chaine definie au niveau zero"
    - a l'entree de fonction : "chaine passee en parametre"
```

```
    - dans le bloc 1 : "chaine definie dans le bloc 1"

Remarque: dans le bloc 2, chaine est un entier valant 12345

$
```

Par défaut, une variable de niveau zéro est globale à tout le programme. Elle est visible dans tous les fichiers où elle est déclarée. De la même façon qu'une fonction, une variable de niveau zéro peut être rendue locale à un fichier au moyen du mot-clé `static`.

L'utilisation de déclarations locales permet d'accroître la modularité d'un programme, en le découpant en autant de fichiers `.c` que de modules logiques. Chaque fichier contient les différentes fonctions du module et les variables que ces fonctions manipulent. Ces variables sont déclarées locales au fichier et sont inaccessibles depuis un autre fichier.

Reconsidérons l'exemple du fichier `caractere_suivant.c` présenté page 66. Selon le principe énoncé précédemment, la variable `caractere_a_lire` doit être rendue locale au fichier. En effet, pour garantir un fonctionnement cohérent de ce module, il est impératif qu'aucune autre fonction du programme ne puisse en modifier la valeur. On déclarera donc cette variable de la façon suivante :

```
static int caractere_a_lire = -1;
```

2.6.6 Définitions et références

Il est important de bien différencier les déclarations correspondant à des **définitions** d'objet et celles correspondant à des **références** à des définitions. Dans le cas d'une fonction, la ligne

⟨*type-de-référence*⟩ ⟨*foncteur*⟩`(...);`

est une déclaration faisant référence à la fonction ⟨*foncteur*⟩. La définition de la fonction s'effectue en rajoutant le bloc d'instructions (voir également 1.4) :

```
⟨type-de-référence⟩  ⟨foncteur⟩(...)
{
    ...
}
```

Dans le cas d'une variable, cette dichotomie n'est pas aussi claire, la même syntaxe pouvant correspondre indifféremment à une définition ou à une référence. On peut par exemple rencontrer la même déclaration de niveau zéro

```
char *argv0;
```

dans plusieurs fichiers différents. Or, il est paradoxal que la définition soit effectuée plusieurs fois. Par conséquent, une seule de ces déclarations doit être considérée comme une définition, et les autres comme des références à cette définition. Cependant, rien ne permet de décider syntaxiquement laquelle de ces déclarations est la définition, et lesquelles sont des références. Cette configuration a été qualifié par le comité de normalisation X3J11 de modèle *lâche de définition/référence*. Il dérive du fonctionnement standard des éditeurs de

liens sous UNIX mais n'est pas totalement portable.

Même en faisant abstraction du problème de la portabilité, d'un strict point de vue méthodologique, il est raisonnable d'expliciter les déclarations en séparant clairement les définitions des références. Pour signaler qu'une déclaration est une référence, on la préfixe par le mot clé `extern`. Par exemple, dans le cas de la variable globale `argv0`, on déclare `char *argv0` dans un seul fichier et `extern char *argv0` dans tous les autres. On utilisera également `extern` pour déclarer une variable définie dans un module de bibliothèque :

```
extern int errno;
```

Un même fichier peut contenir simultanément des références et une définition relatives à un même objet.

Ce modèle a été qualifié de modèle *strict de définition/référence* par le comité de normalisation X3J11. Nous reviendrons sur cet aspect des déclarations de variables globales lorsque nous présenterons les fichiers en-tête en 5.2.2 page 161.

Remarque 9 *Toujours dans le même soucis d'expliciter au maximum les constructions du programme, et bien que ce ne soit pas indispensable, on préfixe également avec le mot-clé* `extern` *les prototypes des fonctions externes (exemple :* `extern int getchar(void);`*).*

2.7 Initialisation de variables

2.7.1 Variables permanentes

Variables élémentaires

Il est possible de spécifier la valeur initiale d'une variable lors de sa déclaration. La syntaxe générale d'une déclaration de variable élémentaire avec initialisation est :

⟨*type-de-référence*⟩ ⟨*construction*⟩ = ⟨*expression*⟩;

Dans le cas des variables permanentes, ⟨*expression*⟩ doit être une expression constante, c'est-à-dire dont la valeur est connue lors de la compilation. Une expression constante est :

- une constante (par exemple 256 ou `'@'`);
- une expression dont les arguments sont des expressions constantes (par exemple `0x01 << 5` ou `&maximum` qui valent respectivement la nombre binaire `100000` et l'adresse de la variable `maximum`).

On trouvera en 3.1.2 et 3.1.3 une présentation détaillée des constantes du langage C.

Voici quelques exemples d'initialisations utilisant des expressions constantes :

```
int   maximum = 256;
char  annulation = 'X';
short etat = 0x01 << 5;
int   *AdresseMaximum = &maximum;
```

On remarquera que dans chaque cas, les types des variables initialisées et des constantes d'initialisation sont cohérents.

Le compilateur traite les déclarations et les initialisations de gauche à droite. Par exemple, la déclaration de niveau zéro

```
char buffer[TAILLE_BUF], *fin_buffer = buffer+TAILLE_BUF-1;
```

provoque à la compilation la suite d'action :

1) réservation d'une zone mémoire de `TAILLE_BUF` octets consécutifs;
2) définition de la constante `buffer` de valeur l'adresse du premier élément de la zone;
3) réservation d'une zone mémoire de la taille d'un pointeur de caractères identifiée par `fin_buffer`;
4) calcul de la constante `buffer+TAILLE_BUF-1` qui est l'adresse du dernier élément du vecteur[5];
5) sauvegarde du résultat dans la variable pointeur `fin_buffer`.

Vecteurs et structures

L'initialisation des objets composés, vecteurs ou structures, s'effectue au moyen de listes de valeurs. Une liste de valeurs a la syntaxe suivante :

{ ⟨*valeur*⟩ , ...}

Par exemple, la déclaration suivante

```
int chiffres_premiers[5] = { 1, 2, 3, 5, 7 };
```

déclare un vecteur de cinq entiers dont le premier élément est initialisé à 1, le second à 2... et le dernier à 7. La taille du vecteur peut être déduite directement du nombre d'éléments de la liste d'initialisation. Par conséquent, il est possible d'écrire simplement :

```
int chiffres_premiers[] = { 1, 2, 3, 5, 7 };
```

en omettant la taille du vecteur. Dans le cas d'un vecteur de caractères,

```
char v1[] = { 'a' , 'b', 'c', 'd', '\0' };
```

peut être abrégé en `char v[] = "abcd";`. Il est important de retenir que les deux déclarations suivantes :

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

ne sont pas équivalentes. La première déclare un vecteur de cinq caractères, initialisé avec les caractères `a`, `b`, `c`, `d` et `\0`; la seconde déclare un pointeur de caractères, initialisé avec l'adresse de la constante chaîne de caractères `abcd`. Il existe une différence fondamentale entre les deux : le contenu du vecteur est modifiable alors que celui de la chaîne ne l'est pas. Généralement, les constantes littérales chaînes de caractères sont rangées dans des blocs de mémoire protégés physiquement en écriture. Toute tentative de modification provoque dans ce cas une erreur.

[5]Cette expression est bien une expression constante puisque `buffer` est une constante créée par le compilateur lors de l'étape 2.

Dans l'exemple suivant, les 4 vecteurs **v1** à **v4** et le pointeur **p** référencent des zones commençant par les quatre caractères **a**, **b**, **c** et **d**.

```
═══════ ini_chn_vct.c ═══════════════════════════
#include <stdio.h>

char v1[] = { 'a', 'b', 'c', 'd' };
char v2[] = "abcd";
char v3[10] = { 'a', 'b', 'c', 'd' };
char v4[10] = "abcd";
char *p = "abcd";

main ()
{
    printf("Taille de v1: %d\n", sizeof v1);
    printf("Taille de v2: %d\n", sizeof v2);
    printf("Taille de v3: %d\n", sizeof v3);
    printf("Taille de v4: %d\n", sizeof v4);

    v2[0] = 'A';
    printf("Modification de v2\n");
    *p = 'A';
    printf("Modification de p\n");
}
═══════════════════════════════ ini_chn_vct.c ═══════
```

Les dimensions de **v1** et **v2** sont déduites implicitement de la spécification d'initialisation. Le taille du vecteur **v2** est celle de la constante **"abcd"**, soit cinq caractères, c'est-à-dire un de plus que celle du vecteur **v1**. Les listes d'initialisation de **v3** et **v4** étant incomplètes, les éléments non spécifiés sont initialisés à zéro. Le pointeur **p** pointe vers une chaîne de même valeur que **v2**, mais dont le contenu n'est pas modifiable. Un tentative de modification provoque une erreur.

```
$ ini_chn_vct
Taille de v1: 4
Taille de v2: 5
Taille de v3: 10
Taille de v4: 10
Modification de v2
Segmentation fault (core dumped)
$ rm core
$
```

Remarque 10 *Cette restriction sur les constantes chaînes de caractères est assez récente et a été définitivement arrêtée dans la norme. On rencontre encore des programmes contenant des constructions de la forme :*

```
char *terminal = "/dev/........";
strcpy(terminal+5, ligne);
```

dans lequel un pointeur est initialisé sur une chaîne, dont la taille est fixée à l'initialisation, et dont le contenu est modifié durant l'exécution du programme.

Un tel programme doit être modifié en

```
char terminal[] = "/dev/........";
...
```

L'exemple suivant montre la déclaration et l'initialisation d'une structure :

```
struct disque
{
    float x, y;
    float rayon;
    int   couleur;
};
typedef struct disque disque;

disque disque_vert = { 0.0, 0.0, 1.0, VERT };
```

On peut naturellement construire des listes de listes; si par exemple on modifie l'exemple précédent en :

```
struct point
{
    float x;
    float y;
};
typedef struct point point;

struct disque
{
    point       centre;
    float       rayon;
    int         couleur;
};
typedef struct disque disque;
```

la déclaration et l'initialisation de l'objet `disque_vert` s'écrivent :

```
disque disque_vert = { {0.0, 0.0}, 1.0, VERT };
```

Voici un autre exemple, concernant la gestion de menus déroulants. Il s'agit de fournir à un gestionnaire de menus la description d'un menu sous la forme d'une liste de choix. Le gestionnaire affiche la liste des choix, récupère la sélection, par exemple au moyen d'une souris, et invoque l'action associée à ce choix. Ici, chaque choix est défini par le libellé du choix et l'adresse de la fonction associée. Le type correspondant s'écrit :

```
struct choix
{
    char *libelle;
    void (*fonction)(void);
};
```

Si on suppose que les fonctions suivantes sont définies de façon externe :

```
extern void message_suivant(void);
extern void message_precedent(void);
extern void repondre(void);
extern void detruire_message(void);
extern void sauver_et_quitter(void)
extern void quitter(void);
```

on peut initialiser un menu par :

```
struct choix menu[] =
{
    { "Affichage du message suivant", message_suivant },
    { "Affichage du message precedent", message_precedent },
    { "Reponse au message courant", repondre},
    { "Destruction du message courant", detruire_message},
    { "Mise a jour et sortie", sauver_et_quitter},
    { "Sortie sans mise a jour", quitter}
};
```

2.7.2 Variables temporaires

Il est également possible d'initialiser les variables temporaires. On a vu qu'une variable temporaire peut être créée et détruite plusieurs fois lors de l'exécution du programme. Par conséquent, contrairement aux variables permanentes, le compilateur ne peut lui affecter de valeur initiale lors de la compilation du programme. En fait, l'initialisation :

```
main()
{
    int nombre_de_fois = 10;

    ...
```

est équivalente à :

```
main()
{
    int nombre_de_fois;

    nombre_de_fois = 10;
    ...
```

Le compilateur génère au début du bloc une instruction effectuant l'affectation de la valeur d'initialisation à la variable à initialiser. Par conséquent, l'initialisation est effectuée **à chaque entrée dans le bloc**. Dans le petit exemple suivant, la fonction `compte_a_rebours` contient une variable temporaire `n` initialisée à la valeur 5 :

```
dynam_ini.c
#define DEBUT 5

void compte_a_rebours();

main()
{
```

```
    compte_a_rebours();
    printf("-> premier essai\n");
    compte_a_rebours();
    printf("-> deuxieme essai\n");
}

void compte_a_rebours()
{
    int n = DEBUT;

    do
    {
        printf("%d ", n);
    }
    while (n--);
}
```
dynam_ini.c

À la différence de l'exemple de la page 67 (fichier `permanente.c`), la variable `n` de la fonction `compte_a_rebours` est initialisée lors de chaque appel.

```
$ dynam_ini
5 4 3 2 1 0 -> premier essai
5 4 3 2 1 0 -> deuxieme essai
$
```

Une variable temporaire peut être initialisée avec une expression quelconque, et non plus uniquement avec une expression constante comme dans le cas des variables permanentes.

L'exemple qui suit combine des initialisations de variables permanentes et temporaires. Il s'agit d'un programme affichant des motifs répétitifs sur sa sortie standard. La fonction `main` utilise deux variables temporaires `nb_lignes` et `nb_colonnes` pour mémoriser le nombre en lignes et en colonnes de motifs à tracer. Le motif est composé d'une alternance des chaînes `motif1` et `motif2`. Ces quatre variables sont initialisées avec des valeurs reçues en arguments de la commande.

La fonction `ligne_suiv` est appelée avec en paramètre la longueur d'une ligne à tracer et les deux motifs. La ligne tracée est composée d'une alternance de ces deux motifs. Les premiers motifs de chaque ligne alternent également. La variable permanente `premier_motif` gère ce mécanisme.

motif.c

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

static void ligne_suiv(int, char*, char*);

main(int argc, char *argv[])
{
    if (argc != 5)
    {
        fprintf(stderr,
                "Usage: %s <nblig> <nbcol> <m1> <m2>\n", argv[0]);
        exit(1);
```

```
    }
    {
        int nb_lignes = 2*atoi(argv[1]);
        int nb_colonnes = atoi(argv[2]);
        char *motif1 = argv[3];
        char *motif2 = argv[4];
        int i;

        printf("\n");
        for (i = nb_lignes; i > 0; i--)
            ligne_suiv(nb_colonnes, motif1, motif2);
        printf("\n");
    }
}

static void ligne_suiv(int nombre_de_motifs, char *m1, char *m2)
{
    static int premier_motif = 0;
    int indice_motif = premier_motif;
    char *motif[] = { m1, m2 };

    while (nombre_de_motifs > 0)
    {
        printf(motif[indice_motif]);
        indice_motif = 1 - indice_motif;
        nombre_de_motifs--;
    }
    printf("\n");
    premier_motif = 1 - premier_motif;
}
```

motif.c

Voici quelques exemples d'exécution de ce programme.

```
$ motif 3 20 /\\ \\/

/\\//\\//\\//\\//\\//\\//\\//\\//\\//\\/
\//\\//\\//\\//\\//\\//\\//\\//\\//\\//\
/\\//\\//\\//\\//\\//\\//\\//\\//\\//\\/
\//\\//\\//\\//\\//\\//\\//\\//\\//\\//\
/\\//\\//\\//\\//\\//\\//\\//\\//\\//\\/
\//\\//\\//\\//\\//\\//\\//\\//\\//\\//\

$
```

```
$ motif 2 30 '()' '  '

()  ()  ()  ()  ()  ()  ()  ()  ()  ()  ()  ()  ()  ()  ()
  ()  ()  ()  ()  ()  ()  ()  ()  ()  ()  ()  ()  ()  ()  ()
()  ()  ()  ()  ()  ()  ()  ()  ()  ()  ()  ()  ()  ()  ()
  ()  ()  ()  ()  ()  ()  ()  ()  ()  ()  ()  ()  ()  ()  ()

$
```

```
$ motif 5 20 '\__' '/  '

\__/  \__/  \__/  \__/  \__/  \__/  \__/  \__/  \__/  \__/
/  \__/  \__/  \__/  \__/  \__/  \__/  \__/  \__/  \__/  \__
\__/  \__/  \__/  \__/  \__/  \__/  \__/  \__/  \__/  \__/
/  \__/  \__/  \__/  \__/  \__/  \__/  \__/  \__/  \__/  \__
\__/  \__/  \__/  \__/  \__/  \__/  \__/  \__/  \__/  \__/
/  \__/  \__/  \__/  \__/  \__/  \__/  \__/  \__/  \__/  \__
\__/  \__/  \__/  \__/  \__/  \__/  \__/  \__/  \__/  \__/
/  \__/  \__/  \__/  \__/  \__/  \__/  \__/  \__/  \__/  \__
\__/  \__/  \__/  \__/  \__/  \__/  \__/  \__/  \__/  \__/
/  \__/  \__/  \__/  \__/  \__/  \__/  \__/  \__/  \__/  \__

$
```

```
$motif 7 15 ___/ \\___

___/\______/\______/\______/\______/\______/\______/\______/
\______/\______/\______/\______/\______/\______/\______/\___
___/\______/\______/\______/\______/\______/\______/\______/
\______/\______/\______/\______/\______/\______/\______/\___
___/\______/\______/\______/\______/\______/\______/\______/
\______/\______/\______/\______/\______/\______/\______/\___
___/\______/\______/\______/\______/\______/\______/\______/
\______/\______/\______/\______/\______/\______/\______/\___
___/\______/\______/\______/\______/\______/\______/\______/
\______/\______/\______/\______/\______/\______/\______/\___
___/\______/\______/\______/\______/\______/\______/\______/
\______/\______/\______/\______/\______/\______/\______/\___
___/\______/\______/\______/\______/\______/\______/\______/
\______/\______/\______/\______/\______/\______/\______/\___

$
```

```
$  motif 3 10 __] [__

__][____][____][____][____][__
[____][____][____][____][____]
__][____][____][____][____][__
[____][____][____][____][____]
__][____][____][____][____][__
[____][____][____][____][____]

$
```

Chapitre 3

Les expressions

Les expressions sont les constructions fondamentales du langage C : il existe quarante-cinq opérateurs d'expression différents. Certaines, comme les expressions arithmétiques et logiques, sont très classiques. D'autres, comme les expressions conditionnelles ou les listes d'expressions, sont moins courantes dans les langages de programmation impératifs. Mais les expressions les plus originales, et également les plus délicates à utiliser, sont les expressions à effet de bord comme les auto-incrémentations et auto-décrémentations, ou encore l'affectation qui est, en C, une expression et non une instruction.

3.1 Expressions élémentaires

Les expressions élémentaires sont les éléments de base intervenant dans la construction des expressions. Elles sont de deux sortes :

- les **constantes littérales** : ce sont des caractères, des nombres entiers ou réels, et des chaînes de caractères; par exemple,

  ```
  -1      1.2323E-1      \0      "syntax error"
  ```

  sont des constantes littérales codant respectivement l'entier -1, le réel 0.2323, le caractère délimiteur de fin de chaîne, et la chaîne formée des caractères `s`, `y`, `n`, `t`, `a`, `x`, ␣, `e`, `r`, `r`, `o`, `r` et `\0`.
- les **identificateurs** : il peut s'agir d'identificateurs de variable, d'identificateurs de fonction, ou d'identificateurs de constante; on parlera dans ce cas de **constantes symboliques**.

3.1.1 Choix des identificateurs

Un **identificateur** est une suite de lettres et de chiffres commençant par une lettre. Le symbole _ est considéré comme une lettre. Le compilateur différencie les lettres minuscules des lettres majuscules. Par conséquent, les quatre mots

```
premierpoint
PREMIERPOINT
Premierpoint
PremierPoint
```

définissent quatre identificateurs différents.

Pour des raisons de lisibilité, nous conseillons d'utiliser de préférence le caractère souligné pour séparer les mots d'un identificateur. Sur l'exemple précédent, on utilisera plutôt les formes :

```
premier_point
Premier_point
PREMIER_POINT
```

D'autre part, nous verrons page 152 qu'il est préférable de réserver les symboles ne comportant que des majuscules et des chiffres aux constructions du préprocesseur. On évitera donc la troisième forme pour définir un identificateur.

Nous utilisons dans cet ouvrage des symboles commençant par une majuscule pour les variables et les types associés à des constantes énumérées, et des symboles en minuscules pour tous les autres identificateurs.

Le choix des identificateurs de variable et de fonction est très important pour la lisibilité des programmes. Il faut tout d'abord respecter les conventions en usage, comme par exemple `argc` et `argv` pour les paramètres de la fonction `main`, ou `i`, `j`, `k` pour les indices de boucle.

Excepté pour ces cas particulier, il faut donner à une variable ou à une fonction un nom complètement explicite. L'habitude consistant à choisir un nom concis accompagné d'un commentaire explicatif :

```
int prmpt;    /* Premier point */
```

est un *tic*, hérité de la programmation en langage d'assembleur, et qu'il faut bannir.

La tendance est actuellement de ne pas se limiter sur la longueur des identificateurs, et il est courant de rencontrer des identificateurs de plusieurs dizaines de caractères. On pourra objecter que la frappe d'identificateurs de grande taille complique la saisie des programmes. Cette objection tombe si on utilise un éditeur disposant d'un mécanisme de complétion dynamique[1]. Il suffit de taper les premiers caractères d'un mot déjà présent dans le *buffer* pour que l'éditeur le complète automatiquement.

La norme ANSI spécifie que les 31 premiers caractères d'un identificateur sont pris en compte par le compilateur. Cette garantie vaut pour les symboles locaux de chaque fichier. Dans le cas de symboles externes, le nombre de caractères significatifs dépend de l'éditeur de liens. Sur ce point, dans un soucis de portabilité, le comité de normalisation a reconduit les conventions minimales en vigueur : seuls les 6 premiers caractères sont significatifs, sans distinction de majuscule et de minuscule. Cette pratique a cependant été qualifiée d'obsolète, ce qui laisse entendre que la norme évoluera vers 31 caractères significatifs pour les symboles internes et externes. En attendant, l'alternative est la suivante : soit écrire des programmes totalement portables mais difficilement lisibles, et donc difficilement maintenables, soit écrire des programmes lisibles mais en limitant leur portabilité. Il est, sauf sur quelques systèmes totalement fermés, généralement possible d'installer un éditeur de lien raisonnable, ce qui favorise le second choix.

Si l'on souhaite cependant s'en tenir à la norme, le comité de normalisation préconise une solution basée sur l'utilisation du préprocesseur (voir chapitre 5) consistant à associer à chaque identificateur externe I un code de six carac-

[1] Sous `emacs`, la complétion dynamique est associée à la clé `M-/`.

tères $C(I)$. Ce code est ajouté en tête du symbole I pour former le nouvel identificateur $C(I)_I$ qui est à la fois explicite et unique. L'utilisation du préprocesseur permet une mise en œuvre relativement souple de cette méthode. Prenons l'exemple d'une variable globale de nom

`sauvegarde_registres_d_etat`

à laquelle est associé le symbole `r1_a03`. La définition de pseudo-constante (voir 5.1.1) suivante

```
#define sauvegarde_registres_d_etat r1_a03_sauvegarde_registres_d_etat
```

permet le remplacement automatique de chaque occurrence de la variable par sa forme unique associée. De cette façon, les identificateurs restent lisibles même après la phase de macro-expansion. En ordonnant toutes ces définitions dans un fichier en-tête unique, et en définissant une stratégie globale d'attribution des codes, il est possible de contrôler qu'un même code n'est pas associé à deux identificateurs.

3.1.2 Constantes littérales

Constantes entières

Une constante entière est un mot écrit avec des chiffres et éventuellement certaines lettres (`x`, `a`, `b`, `c`, `d`, `e` et `f`) majuscules ou minuscules. Une constante numérique est **décimale**, c'est-à-dire exprimée en base 10, si elle ne contient que des chiffres et si elle commence par un caractère différent de zéro :

`10240`

Une constante commençant par un `0` et ne contenant que les caractères `0`, `1`, `2`, `3`, `4`, `5`, `6` et `7` est une constante **octale**, c'est-à-dire exprimée en base 8. Par exemple,

`0177`

a pour valeur 127 en base 10.

Remarque 11 *Les compilateurs* C *traditionnels acceptent généralement des constantes octales contenant les caractères* `8` *et* `9`*; par exemple, la constante* `029` *est convertie en 25 en base 10* $(2 \times 8 + 9)$*. Cette écriture est incorrecte en* C *standard.*

Une constante commençant par les deux caractères `0x` ou `0X` est une constante **hexadécimale**, c'est-à-dire exprimée en base 16. Elle s'écrit au moyen des chiffres et des lettres de `a` à `f`; les lettres peuvent être indifféremment des majuscules ou des minuscules. Les lettres de `a` à `f` codent les chiffres hexadécimaux de 10 à 15. Par exemple, la valeur décimale 255 peut s'écrire :

`0xff`

En effet, la lettre `f` codant le chiffre hexadécimal 15, la valeur de `0xff` est

$$15 \times 16^1 + 15 \times 16^0 = 255$$

Une constante entière est par défaut de type `int`, c'est-à-dire `short` ou `long` suivant les implémentations (voir 2.1.1). Une constante est explicitement de type `long` dans deux cas :

- sa valeur est supérieure à la valeur maximale d'un entier court, signé dans le cas d'une constante décimale, et sans signe sinon (par exemple 32768 ou 0x10000);
- elle est terminée par le caractère `l` ou `L` (par exemple 1L ou -256L).

Le suffixe `U` (ou `u`) désigne le type `unsigned`; il peut être combiné avec le suffixe L :

`65535U        1Lu        1UL`

Remarque 12 *Les expression* `+3` *et* `-3` *ne sont pas des constantes littérales, mais des expressions constantes, construites avec un opérateur unaire (respectivement* `+` *et* `-`*) et la constante entière* `3`*. Il n'existe pas en C traditionnel d'opérateur unaire* `+`*; l'expression +3 est donc dans ce cas incorrecte.*

Constantes réelles

Une constante réelle est un mot écrit avec des chiffres, et les caractères `.`, `-`, et `E` ou `e`. Elle doit comporter au moins un *point* ou un `e`, et au plus un de chaque :

`37.2        0.        1E-10        2.99792e6`

La valeur d'une constante de la forme $x\mathrm{E}n$, ou $x\mathrm{e}n$ est $x \times 10^n$. Les suffixes `F` (ou `f`) et `L` (ou `l`) appliqués sur une constante réelle spécifient des constantes de type *réel simple précision* et *réel précision étendue* :

`1.0F        0.0L`

Constantes caractère

Une constante caractère est :

- un caractère entre guillemets simples : `'A'`, `'+'`, etc.
- une suite de deux caractères entre guillemets simples, dont le premier est le caractère \ (voir tableau 3.1) : `'\n'`, `'\f'`, etc.
- un mot de la forme `'\`*nnn*`'`, *nnn* étant le code du caractère exprimé en base 8 : `'\033'`, `'\177'`, etc.
- un mot de la forme `'\x`*nn*`'`, *nn* étant le code du caractère exprimé en base 16 : `'\xff'`, `'\x0b'`, etc.

Le préfixe L permet de désigner des constantes caractère étendu (voir 2.1.3) :

`L' '        L'\a'        L'\033'`

Le code `\a` est un caractère d'alerte, correspondant en général au caractère `BELL`, de code ASCII 7, qui provoque l'émission d'un signal sonore par le terminal. Le code \? représente le point d'interrogation. Il a pour but d'éliminer certaines ambiguïtés provenant de l'introduction dans la norme ANSI des **trigraphs**. Il s'agit de suites de trois caractères codant les 9 caractères non standard[2] de la table ASCII. La liste des trigraphs est donnée dans le tableau 3.2.

---

[2]L'ensemble des codes ASCII standard sur sept bits, défini par la norme *ISO 646-1983*, ne contient pas les caractères `#`, `[`, `\`, `]`, `^`, `{`, `|`, `}`et `~`.

La substitution d'un trigraph par le caractère équivalent est effectuée avant l'appel au préprocesseur (voir chapitre 5). Considérons par exemple le programme suivant :

```
═══════ trigraph.c ══════════════════════════
??=define MESSAGE "??/n??/tDes trigraphs\?\?!??/n??/n"

main()
??<
   printf("%s", MESSAGE);
??>
══════════════════════════════ trigraph.c ═══════
```

Il peut être compilé avec gcc de la façon suivante :

```
$ gcc -g -ansi trigraph.c -o trigraph
$ trigraph

        Des trigraphs??!

$
$  gcc -ansi -E trigraph.c
# 1 "trigraph.c"

main()
{
   printf("%s", "\n\tDes trigraphs\?\?!\n\n" );
}
$
```

L'option -E utilisée lors de la seconde invocation à gcc permet de visualiser le résultat produit par le préprocesseur (voir 5.1.1), et donc après la phase de traduction des trigraph.

Le code d'échappement \? peut être nécessaire pour coder une suite de plusieurs points d'interrogation dans une chaîne. Un programme écrit en C traditionnel et contenant plusieurs points d'interrogation consécutifs suivis d'un des caractères =, (, /,), ', <, !, > et -, comme par exemple la ligne

```
#define MESSAGE "\n\tDes trigraphs??!\n\n"
```

n'est donc pas compatible avec la norme Ansi.

Une constante caractère a pour valeur le code de ce caractère en machine; il s'agit en général, du code Ascii[3]. Par exemple, la constante 'A' a pour valeur le code du caractère A, (65 en Ascii), et '\033' code le caractère de valeur 27, c'est-à-dire le caractère Ascii ESCAPE. La table des codes Ascii des caractères est donnée en annexe C.

[3]Il existe d'autres codages, comme le codage EBCDIC. Pour être totalement portable, un programme ne doit faire aucune supposition sur le codage caractères. Cependant, la plupart des environnements actuels sont construits sur le codage ASCII ce qui permet raisonnablement de considérer celui-ci comme le codage standard.

Constante	Valeur	Signification courante
\a		alerte sonore
\b	BS	retour arrière (caractère *"backspace"*)
\f	FF	fin de page (caractère *"form feed"*)
\n	NL	fin de ligne (caractère *"line feed"*)
\r	CR	retour chariot (caractère *"carriage return"*)
\t	HT	tabulation horizontale
\v	VT	tabulation verticale
\\\\	\\	caractère d'échappement
\\'	'	apostrophe
\\"	"	guillemet
\\?	?	point d'interrogation
\x*nn*		caractére de code hexadécimal *nn*
nnn		caractére de code octal *nnn*

Tableau 3.1: Caractères spéciaux

Trigraph	Caractère équivalent
??=	# (*0x23*)
??(	[(*0x5b*)
??/	\ (*0x5c*)
??)	] (*0x5d*)
??'	^ (*0x5e*)
??<	{ (*0x7b*)
??!	| (*0x7c*)
??>	} (*0x7d*)
??-	~ (*0x7e*)

Tableau 3.2: Liste des *trigraphs*

Chaînes de caractères

Une constante chaîne est une suite de caractères entre guillemets doubles (caractère "). Toutes les formes de définition de caractères sont utilisables.

Une chaîne de caractères est implémentée sous la forme d'une suite de caractères terminée par le caractère \0. Sa valeur est l'adresse mémoire de son premier caractère, et son type est *pointeur de caractères.*

Par exemple, la constante

```
"\007Chaine qui \"sonne\"\n"
```

est formée de la suite des 20 caractères

```
🔔Chaine␣qui␣"sonne"↵
```

terminée par le caractère \0 (le symbole 🔔représente le caractère BELL). Le caractère L placé devant le premier guillemet spécifie une chaîne de caractères

étendus (exemple : `L"\xf00a\xf00b"`).

Une chaîne dont le dernier caractère sur une ligne est \ se continue sur la ligne suivante. Le programme suivant illustre l'utilisation de caractères spéciaux pour construire une chaîne.

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

char *chaine = "\
\n\
\t/-------------------------------------------\\\n\
\t| Pour ecrire une chaine sur plusieurs lignes, |\n\
\t|   il suffit de terminer chaque ligne par \\   |\n\
\t\\-------------------------------------------/\n\n";

main()
{
    printf(chaine);
}
longuechaine.c
```

Les constantes `'\n'` et `'\t'` permettent de gérer les passages à la ligne et les indentations. Le caractère \ est obtenu au moyen de la constante `'\\'`.

```
$ longuechaine

        /----------------------------------------------\
        | Pour ecrire une chaine sur plusieurs lignes, |
        |   il suffit de terminer chaque ligne par \   |
        \----------------------------------------------/

$
```

Il est possible d'écrire plusieurs constantes chaînes de caractères les unes à la suite des autres; elles sont automatiquement concaténées par le compilateur :

```
chaines_concat.c
main()
{
    printf("- Premiere ligne: quelques mots...\n"
           "- Seconde ligne: cette ligne est "
           "plus longue...\n");
}
chaines_concat.c
```

Cela permet de présenter les programmes de façon plus lisible, sans que l'indentation des lignes dans le programme source ne perturbe l'affichage du résultat.

```
$ chaines_concat
- Premiere ligne: quelques mots...
- Seconde ligne: cette ligne est plus longue...
```

```
$
```

Cela permet également de résoudre certaines ambiguïtés dans la définitions des constantes chaînes. Par exemple, `"\x7F" "DEL"` est en ASCII la chaîne composée des quatre caractères `DEL`, `D`, `E` et `L`. Par contre la chaîne `"\x7FDEL"` est une chaîne de 2 caractères dont le premier est l'octet de poids faible de la valeur hexadécimale `0x7fde`, et le second est le caractère `L`. En effet, dans une chaîne, le décodage d'une constante caractère hexadécimale se poursuit jusqu'au premier caractère non hexadécimal rencontré. Le décodage de cette chaîne par le compilateur provoque d'ailleurs un message d'erreur :

```
t.c:3: warning: escape sequence out of range for character
```

Remarque 13 *Une chaîne de caractères définie au moyen de guillemets est une constante; par conséquent, il n'est pas possible d'en modifier le contenu.*

3.1.3 Constantes symboliques

Constantes énumérées

Une énumération de constantes symboliques se définit au moyen de la construction **enum**. Nous allons en donner seulement un exemple, sa syntaxe ayant déjà été définie en 2.1.2. La déclaration

```
enum Chateau
{
    Haut_Brion,
    Margaux,
    Latour,
    Cheval_Blanc,
    Pape_Clement
};
```

définit les cinq constantes symboliques `Petrus`, `Haut_Brion`, `Margaux`, `Latour`, `Cheval_Blanc` et `Pape_Clement`.

Identificateurs de vecteurs

Un **vecteur** est une suite d'éléments tous de même type (vecteur d'entiers, vecteur de pointeurs, vecteur de vecteurs...). Un **identificateur de vecteur** est une constante symbolique dont la valeur est l'adresse du premier élément du vecteur. Si le type d'un élément est $\langle t \rangle$, le type de l'identificateur est *pointeur sur un objet de type* $\langle t \rangle$.

Le programme `ident_vect.c` illustre la différence entre la valeur d'un symbole de vecteur, qui est un pointeur, et celle de ses éléments.

```
==== ident_vect.c ====
#include <stdio.h>

main()
```

```
{
    int vct_entiers[] = {2, 4, 6, 8, 0};

    printf("\nL'expression \"vct_entiers\" vaut 0x%p;\n",
          vct_entiers);
    printf(" c'est l'adresse memoire du debut du tableau.\n");
    printf("\nL'expression \"vct_entiers[0]\" vaut %d;\n",
          vct_entiers[0]);
    printf(" c'est la valeur du premier element.\n\n");
}
```

ident_vect.c

La spécification de format %p permet de faire afficher la valeur d'un pointeur en notation hexadécimale.

```
$ ident_vect

L'expression "vct_entiers" vaut 0xf7fff900;
 c'est l'adresse memoire du debut du tableau.

L'expression "vct_entiers[0]" vaut 2;
 c'est la valeur du premier element.

$
```

Le programme suivant est incorrect car l'identificateur v est utilisé en partie gauche d'une affectation.

err_aff_vecteur.c

```
main()
{
    char v[16];

    v = 'c';
}
```

err_aff_vecteur.c

Sa compilation avec gcc provoque le message d'erreur

```
$ gcc -g -c err_aff_vecteur.c
err_aff_vecteur.c: In function 'main':
err_aff_vecteur.c:5: incompatible types in assignment
$
```

et avec le compilateur traditionnel cc

```
$ cc -g -c err_aff_vecteur.c
"err_aff_vecteur.c",line 5: illegal lhs of assignment operator
"err_aff_vecteur.c",line 5: warning: illegal combination of
                            pointer and integer, op =
```

Le mot lhs est une abréviation de *left handside*; ce dernier message signale

que le membre gauche de l'affectation est incorrect.

Identificateurs de fonctions

Un identificateur de fonction est une constante de type pointeur de fonctions; sa valeur est l'adresse de la première instruction de la fonction. Le programme suivant est l'analogue pour les fonctions de `ident_vect.c`.

```
======== ident_fonc.c ========
#include <stdio.h>

long fcnt();

main()
{
    printf("\nL'expression \"fcnt\" vaut 0x%p;\n", fcnt);
    printf("  c'est l'adresse memoire de la fonction.\n");
    printf("\nL'expression \"fcnt()\" vaut %d;\n", fcnt());
    printf("  c'est le resultat de la fonction.\n\n");
}

long fcnt()
{
    return 56846091;
}
======== ident_fonc.c ========
```

On distingue ici la valeur du pointeur de fonction de celle de son évaluation.

```
$ ident_fonc

L'expression "fcnt" vaut  0x23a0;
  c'est l'adresse memoire de la fonction.

L'expression "fcnt()" vaut 56846091;
  c'est le resultat de la fonction.

$ nm ident_fonc | grep fcnt
000023a0 T _fcnt
$
```

La commande NM(1) donne l'adresse en mémoire des variables statiques d'un programme exécutable. On retrouve bien la valeur hexadécimale `0x23a0`.

Un identificateur de fonction est défini explicitement dans une déclaration ou une définition de fonction :

```
extern int lire_entier(void);

char *lire_mot(char *message)
{
    ...
}
```

Il peut également être défini implicitement dans une expression : lorsque le

compilateur C rencontre une expression de la forme

⟨*ident*⟩`(...)`

si ⟨*ident*⟩ n'a pas été déclaré auparavant, il considère ⟨*ident*⟩ comme un identificateur de fonction entière.

Le mécanisme de déclaration implicite d'une fonction dans le type `int` *était très utilisé dans les premiers programmes C pour mettre en œuvre des procédures et des fonctions entières. En C* ANSI, *dans* **chaque** *fichier, il faut s'imposer de déclarer explicitement* **chaque** *fonction utilisée au moyen de son prototype, y compris s'il s'agit d'une fonction entière. Les procédures sont déclarées de type* `void`.

Les déclarations

```
int lire_caractere(void);
int (*carsuiv)(void) = lire_caractere;
```

déclarent respectivement `lire_caractere`, un identificateur de fonction retournant un entier, et `carsuiv`, un pointeur de fonctions sans paramètre retournant un entier, initialisé avec l'adresse de la fonction `lire_caractere`. Sous ces hypothèses, les deux appels de fonction `lire_caractere()` et `carsuiv()` sont équivalents[4].

3.1.4 Variables et g-valeurs

Une **variable** est une zone mémoire modifiable par une affectation. Elle est caractérisée par son identificateur et son type, ce dernier guidant l'interprétation de son contenu.

Une **variable élémentaire** est un objet de base du programme. Son type est, soit un type de base (caractère, entier ou réel), soit un type défini par énumération de constantes, soit encore un type *pointeur d'objets*[5]. Les autres variables sont des **variables structurées**, définies avec les opérateurs `struct` et `union`.

En C, un vecteur n'est pas une variable mais une **liste de variables**. *Il n'est, en effet, pas directement modifiable au moyen d'une affectation. Par contre, il est possible de modifier chacun de ses éléments.*

La valeur d'une variable peut dépendre de son utilisation dans une expression. Considérons par exemple l'affectation `y = y + 1` où `y` est une variable entière. Lors de l'évaluation de `y + 1` c'est le contenu de `y` qui est utilisé; par contre, lors de l'évaluation du membre gauche, c'est sa référence. L'expression `y + 1 = y` est illégale parce que l'expression `y + 1` ne fait référence à aucun objet, ce qui rend l'affectation impossible.

Si `pnt_car` est un pointeur de caractères, l'expression `*pnt_car = 'a'` est correcte; en effet, `*pnt_car` fait référence à un objet qui est le caractère pointé

[4]Certains compilateurs n'acceptent pas la seconde forme (voir remarque 14 page 102).

[5]La taille d'une variable élémentaire d'un type donné peut varier d'une implémentation à l'autre; cependant, elle est toujours inférieure ou égale à une taille maximale, dépendant elle aussi de l'implémentation pouvant aller en général de 4 à 16 octets. Cette taille maximale est liée à la taille des registres de la machine.

par `pnt_car`. C'est également le cas de l'expression `*(pnt_car + 1)` qui fait référence au caractère suivant le caractère pointé par `pnt_car`.

Il est nécessaire de différencier les expressions faisant référence à une variable des autres expressions. Dans la terminologie anglo-saxonne, on appelle les premières des **lvalues**. C'est une abréviation de *left values*, c'est-à-dire *"valeurs pouvant être utilisées dans la partie gauche d'une affectation"*. Par analogie, nous les appellerons des **g-valeurs**. Si on considère la liste de déclarations suivante

```
float echelle;
float coefficients[NBCOEFF];
struct Client
{
    ...
    int age;
    ...
};
struct Client client;
```

les expressions

```
echelle
coefficients[NBCOEFF - 1]
*coefficients
client
client.age
```

sont toutes des g-valeurs. Par contre, `coefficients` n'en est pas une.

En utilisant les conversions explicites de types (voir 3.3.2), il est possible de former des g-valeurs sans utiliser d'identificateur; par exemple, l'expression

```
*((char *) 0xc0000000)
```

est une g-valeur référençant l'octet d'adresse `0xc0000000`. Sur un micro-ordinateur, une telle expression permet d'accéder à une adresse physique[6].

3.2 Mécanismes de construction

3.2.1 Constructions d'expressions

Une **expression** est une construction réalisée au moyen d'expressions élémentaires et d'**opérateurs**. Les arguments d'un opérateur sont appelés ses **opérandes**. Par exemple, dans l'expression `x + (2*y)`, l'opérateur `+` possède deux opérandes qui sont les sous-expressions `x` et `(2*y)`. Une expression correcte doit respecter diverses règles de construction. Ces règles portent essentiellement sur :

– le nombre d'opérandes qu'accepte un opérateur, encore appelé son **arité**;
– les types autorisés pour chaque opérande.

Il existe plusieurs façons d'utiliser les opérateurs dans une construction. Un opérateur d'arité un, ou opérateur **unaire** peut être **préfixe**, c'est-à-dire

[6] Dans le cas d'un processus UNIX, cette adresse n'est pas une adresse physique, mais une adresse logique relative à l'espace mémoire du processus.

placé avant son opérande, ou **postfixe**, c'est-à-dire placé après. Par exemple, le *moins unaire* est un opérateur préfixe, car on écrit `-3` et non `3-`. Certains opérateurs unaires du langage C, comme la post-incrémentation, sont des opérateurs postfixes. Les opérateurs d'arité deux sont **infixes**, c'est-à-dire placés entre leurs deux opérandes.

Le mécanisme de construction d'expression est récursif : on définit une expression complexe à partir de sous-expressions plus simples, les objets de base étant les constantes littérales et les identificateurs.

Ces constructions sont définies dans les manuels de référence des langages de programmation sous la forme de *diagrammes syntaxiques*, ou de *règles de grammaire.* C'est cette seconde forme qui est adoptée dans le manuel de référence du langage C. On trouve par exemple les définitions suivantes :

expression-additive : *expression-additive* + *expression-multiplicative*
expression-additive - *expression-multiplicative*
expression-égalité : *expression-égalité* < *expression-relation*
expression-égalité <= *expression-relation*

Les règles ci-dessus sont directement extraites de la grammaire du langage C standard donnée dans l'ouvrage de référence de Kernighan et Ritchie. Cette grammaire définit de façon non ambiguë la structure syntaxique du langage. Dans cet ouvrage, nous adopterons une forme moins précise mais plus intuitive; par exemple, à la place des deux premières règles, nous écrirons :

expression-additive : *expression* + *expression*
expression - *expression*

Ces règles, qui définissent la syntaxe des opérateurs + et -, doivent être lues de la façon suivante : si $\langle e_1 \rangle$ et $\langle e_2 \rangle$ sont deux expressions correctes alors les expressions

$$\langle e_1 \rangle + \langle e_2 \rangle$$
$$\langle e_1 \rangle - \langle e_2 \rangle$$

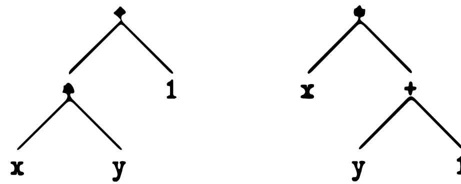
le sont également.

Toute expression correcte possède un type et une valeur. Par exemple, si $\langle e_1 \rangle$ et $\langle e_2 \rangle$ sont deux expressions correctes de type entier (resp. réel), l'expression $\langle e_1 \rangle + \langle e_2 \rangle$ est également de type entier (resp. réel), et sa valeur est la somme dans $\mathbb{N}$ (resp. dans $\mathbb{R}$) des valeurs respectives de $\langle e_1 \rangle$ et $\langle e_2 \rangle$.

3.2.2 Priorité des opérateurs

L'écriture linéaire des expressions est ambiguë. Par exemple, l'expression `x*y+1` peut être décomposée de deux façons différentes, qui sont `(x*y)+1` et `x*(y+1)`

Dans le premier cas, `x*y+1` est la somme des deux sous-expressions `x*y` et `1`; dans le second, elle est le produit de `x` et de `y+1`. Il est naturel de représenter une expression sous la forme d'un arbre, appelé **arbre de syntaxe**. Les deux arbres de syntaxe qu'il est possible d'associer à l'expression `x*y+1` sont :



La première possibilité correspond à l'utilisation habituelle de l'addition et de la multiplication; on dit que * est **prioritaire** par rapport à +. On remarque que, contrairement à l'écriture linéaire, l'arbre de syntaxe n'est pas ambigu.

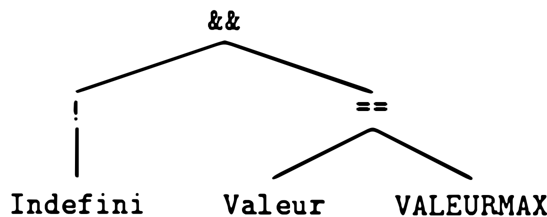
Les opérateurs de construction d'expressions du langage C sont regroupés selon quinze **niveaux de priorité**. Le tableau 3.3 donne la liste des opérateurs par ordre de priorité décroissante. Les opérateurs figurant sur une même ligne sont de même niveau de priorité. Ce tableau permet de déterminer sans ambiguïté la structure syntaxique, et par conséquent la valeur de n'importe quelle expression bien formée du langage. Considérons par exemple l'expression

```
! Indefini && Valeur == VALEURMAX
```

où `!`, `&&` et `==` représentent respectivement l'opérateur unaire de négation, le connecteur logique *et*, et le test d'égalité. On peut l'interpréter de cinq façons différentes :

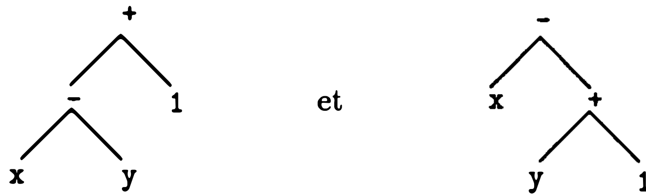
```
(1)    ((!Indefini) && Valeur) == VALEURMAX
(2)    (!(Indefini && Valeur)) == VALEURMAX
(3)    !(Indefini && (Valeur == VALEURMAX))
(4)    !((Indefini && Valeur) == VALEURMAX)
(5)    (!Indefini) && (Valeur == VALEURMAX)
```

On peut lire dans le tableau 3.3 que l'opérateur `!` est prioritaire par rapport à `&&`, ce qui élimine les interprétations *(2)*, *(3)* et *(4)*. On y voit également que `==` est prioritaire par rapport à `&&`, ce qui élimine la première interprétation. L'arbre de syntaxe de cette expression est donc celui correspondant à l'expression *(5)* :



3.2.3 Associativité des opérateurs

Lorsque deux opérateurs ont le même niveau de priorité, il subsiste une ambiguïté. Par exemple, l'expression `x-y+1` peut être construite de deux façons différentes, `(x-y)+1` et `x-(y+1)`, dont les arbres de syntaxe respectifs sont :



Dans le premier cas, les opérateur sont dits **associatifs de gauche à droite**, car l'opérateur le plus à gauche est appliqué en premier; ils sont dits **associatifs de droite à gauche** dans le second cas (les opérateurs + et - sont usuellement associatifs de gauche à droite). On trouvera le mode d'associativité de chaque opérateur dans le tableau 3.3, GD signifiant *associatif de gauche à droite* et DG *associatif de droite à gauche.*

<table>
<tr><th>Opérateur</th><th>Associativité</th><th>Arité</th></tr>
<tr><td>() []
-> .</td><td>GD</td><td>–
2</td></tr>
<tr><td>! ~ ++ -- -
(⟨type⟩) * & sizeof
+ (ANSI)</td><td>DG</td><td>1</td></tr>
<tr><td>* / %</td><td>GD</td><td>2</td></tr>
<tr><td>+ -</td><td>GD</td><td>2</td></tr>
<tr><td><< >></td><td>GD</td><td>2</td></tr>
<tr><td>< <= > >=</td><td>GD</td><td>2</td></tr>
<tr><td>== !=</td><td>GD</td><td>2</td></tr>
<tr><td>&</td><td>GD</td><td>2</td></tr>
<tr><td>^</td><td>GD</td><td>2</td></tr>
<tr><td>|</td><td>GD</td><td>2</td></tr>
<tr><td>&&</td><td>GD</td><td>2</td></tr>
<tr><td>||</td><td>GD</td><td>2</td></tr>
<tr><td>?:</td><td>DG</td><td>3</td></tr>
<tr><td>= += -= *= /= %=
>>= <<= &= ^= |=</td><td>DG</td><td>2</td></tr>
<tr><td>,</td><td>GD</td><td>2</td></tr>
</table>

Tableau 3.3: Opérateurs d'expression rangés par priorité décroissante

3.2.4 Type des opérandes

Le type d'une expression est statique : on peut le déterminer en lisant cette expression dans le programme source et il ne dépend pas de l'exécution du programme. Le type d'une expression se déduit du type de ses sous-expressions et de son opérateur.

Un même opérateur peut correspondre, selon le type de ses opérandes, à plusieurs opérations différentes. Par exemple, l'opérateur + peut désigner l'ad-

dition de deux entiers, ou de deux réels, selon que ses opérandes sont des expressions de type entier ou réel. Il existe également une addition entre pointeur et entier, c'est-à-dire entre une *adresse* et un *décalage.*

Dans une construction où les types des opérandes ne correspondent pas à une configuration acceptée par l'opérateur, on a le choix entre interdire la construction ou effectuer, lorsque c'est possible, une conversion automatique de type sur l'un des deux opérandes. C'est cette seconde option qui a été retenue en C. On parle dans ce cas de **conversion implicite** de type.

L'utilisation d'opérateurs acceptant plusieurs types d'opérande n'est pas adoptée dans tous les langages. Par exemple dans le langage PASCAL, il existe deux opérateurs de division distincts, un pour la division entière, l'autre pour la division réelle. A l'opposé, d'autres langages, comme les langages ADA ou C++, intègrent totalement ce principe; ce mécanisme est appelé la *surcharge d'opérateur.*

Il est nécessaire de disposer de mécanismes de **conversion explicite** de type. Ce mécanisme doit exister pour tous les types possibles, que ce soient les types de base ou les types construits. Deux options sont possibles : considérer la conversion de type comme une fonction (c'est par exemple le cas en ADA), ou comme un opérateur, ce qui est le cas du langage C.

On construit un opérateur de conversion explicite de type en plaçant le nom du type entre parenthèses. L'opérateur ainsi obtenu est unaire et préfixe. Par exemple, si `total` et `quantite` sont deux variables entières, l'expression

```
(float) total / (float) quantite
```

calcule une division réelle entre les valeurs respectives de ces variables, préalablement converties en réels.

3.3 Syntaxe et sémantique des opérateurs

Nous allons maintenant présenter dans le détail les différents opérateurs d'expressions du langage C. Nous noterons e la valeur de l'expression $\langle e \rangle$.

3.3.1 Expressions primaires

Une **expression primaire** est de l'une des sept formes suivantes :

expression-primaire : *identificateur*
constante-littérale
(*expression*)
expression-primaire . *identificateur*
expression-primaire -> *identificateur*
expression-primaire [*expression*]
expression-primaire (*liste-d'expressions*)

Nous avons déjà présenté les deux premières formes en 3.1. La troisième est un parenthésage, permettant de modifier les règles de priorité des opérateurs. La valeur et le type d'une telle expression sont ceux de la sous-expression enfermée entre les parenthèses.

La quatrième forme est une **sélection de champ**; elle permet d'accéder aux champs d'une structure ou d'une union. Si une expression $\langle e \rangle$ est de la

forme $\langle e_1 \rangle$`.`$\langle ident \rangle$ alors $\langle e_1 \rangle$ doit être une structure ou une union, et $\langle ident \rangle$ l'identificateur de l'un de ses champs. Le type et la valeur de $\langle e \rangle$ sont ceux du champ $\langle ident \rangle$. Soit par exemple la structure

```
struct lexeme
{
    int code;
    char *identificateur;
};
```

et la variable `lexeme` de type `struct lexeme`. Alors l'expression `lexeme.code` est de type entier et a pour valeur le contenu du champ `code`. Si `pntlex` est un pointeur de structure `struct lexeme`, alors l'expression `*pntlex` est un objet de type `struct lexeme`. Par conséquent, `(*pntlex).code` est également une expression correcte. Cette dernière expression peut être abrégée en `pntlex->code` qui est la cinquième forme d'expression primaire.

L'expression

$$\langle e_1 \rangle [\langle e_2 \rangle]$$

est correcte si $\langle e_1 \rangle$ est un pointeur différent d'un pointeur de fonctions (par exemple un identificateur de vecteur ou de pointeur), et si $\langle e_2 \rangle$ est une expression entière. Si $\langle e_1 \rangle$ est un pointeur de type t et contenant une adresse α, et si la valeur de $\langle e_2 \rangle$ est un entier n, alors la valeur de

$$\langle e_1 \rangle [\langle e_2 \rangle]$$

est celle du $n + 1^{\text{ème}}$ objet de type t à partir de l'adresse α. Par exemple, compte tenu des déclarations suivantes :

```
int   position;
char  nom_fichier[TAILLE_NOM];
char  *lettre = nom_fichier;
```

les expressions

```
nom_fichier[0]
nom_fichier[position - 1]
lettre[position/2]
```

sont trois expressions correctes de type caractère. La valeur de

```
nom_fichier[0]
```

est le premier caractère du vecteur de caractères `nom_fichier`. Remarquons qu'une expression peut être syntaxiquement correcte et conduire à une erreur lors de son évaluation. Par exemple, l'expression `nom_fichier[position-1]` n'est sémantiquement correcte que si la valeur de la variable `position` est comprise entre 1 et le nombre d'éléments du vecteur. Dans le cas contraire, il se produira une erreur à l'exécution lors de l'évaluation de l'expression.

La dernière forme d'expression primaire est

$$\langle e \rangle (\langle e_1 \rangle, \ldots, \langle e_n \rangle)$$

Cette expression, qui réalise un **appel de fonction**, est correcte si $\langle e \rangle$ est de type pointeur de fonctions, par exemple un identificateur de fonction. Les expressions $\langle e_1 \rangle$, $\langle e_2 \rangle$, ..., $\langle e_n \rangle$ sont les **paramètres effectifs** de l'appel; ils sont évalués et les valeurs obtenues sont transmises à la fonction. Il s'agit par conséquent d'un mécanisme d'**appel par valeur**. Le type et la valeur

d'un appel de fonction sont respectivement le type de la fonction, et la valeur qu'elle retourne.

L'ordre d'évaluation des paramètres est indéterminé. Considérons par exemple l'expression :

```
Ranger(Base, max-1, lire_chaine("-> "))
```

Un schéma d'évaluation possible est le suivant :

1) évaluation des trois sous-expressions constituant les paramètres de l'appel à `Ranger`; cette évaluation est effectuée dans un ordre indéterminé pouvant varier selon les compilateurs utilisés, par exemple :
   a) évaluation de l'expression `lire_chaine("-> ")` :
      (i) évaluation du paramètre : le résultat est l'adresse α de la chaîne de caractères `"-> "`;
      (ii) appel de la fonction `lire_chaine` avec en paramètre l'adresse α; cette fonction retourne une valeur v_1;
   b) évaluation de l'expression `max-1`; le résultat est une valeur v_2;
   c) évaluation de l'expression élémentaire `Base` : le résultat est une valeur v_3;
2) appel de la fonction `Ranger` avec comme paramètres effectifs les trois valeurs v_3, v_2, v_1.

Remarque 14 *Certaines implémentations non standard du langage C n'acceptent pas d'expression de la forme* $\langle pf \rangle$`()`, *où* $\langle pf \rangle$ *est un pointeur de fonctions. Il est alors nécessaire d'écrire* `(*`$\langle pf \rangle$`)()`.

3.3.2 Conversion de types

Conversions implicites

Certains opérateurs, principalement les opérateurs arithmétiques et logiques et les opérateurs d'affectation, peuvent provoquer des conversions implicites de type lors de leur évaluation. Ces conversions sont effectuées lorsque les types des opérandes sont différents, mais comparables. Les types de base sont ordonnés de la façon suivante :

$$\texttt{void}$$

$$\texttt{char} < \texttt{short} \leq \texttt{int} \leq \texttt{long} < \begin{cases} \texttt{float} < \texttt{double} < \texttt{long double} \\ \texttt{void*} < \textit{pointeur sur} \end{cases}$$

Les énumérations de constantes sont équivalentes au type `int`. Le type `void` étant vide, il n'est comparable avec aucun autre. Tous les types *"pointeur sur"* sont au même niveau :

- supérieurs à `void *`,
- incomparables entre eux.

Le type `unsigned long` est toujours supérieur à `long`.

Dans le cas des expressions arithmétiques et logiques, les conversions implicites sont effectuées du type le plus faible vers le type le plus fort. Lors d'une affectation, le résultat du membre de droite est converti dans le type du membre de gauche. Par exemple, si `car` est une variable de type caractère,

lors de l'évaluation de l'expression `car = 'A' + 1` la constante `'A'` est convertie en la valeur entière 65 au moment de l'évaluation de l'addition, puis le résultat de l'addition est reconverti en un caractère, ici le caractère `'B'`, lors de l'évaluation de l'affectation.

Dans un souci de lisibilité, on essaiera d'éviter les conversions implicites. Par exemple, si `car_nul` et `debit` sont respectivement un caractère et un réel, on écrira

```
debit = 0.0
car_nul = '\0'
```

de préférence à `debit = 0` et `car_nul = 0`.

Conversions explicites

Les conversions explicites de types s'effectuent au moyen d'opérateurs de conversion. Un opérateur de conversion de type se construit à partir d'un nom de type et des symboles de parenthèses. La syntaxe d'une expression de conversion de type est la suivante :

expression-conversion : (*type*) *expression*

type pouvant être un identificateur de type ou une expression de type (voir 2.5.2). La valeur de l'expression ainsi construite est le résultat de la conversion de *expression* dans le type *type*. Par exemple, la valeur de l'expression `(float) 3` est le réel 3.0, et celle de l'expression `(int) 5.1287e2` est l'entier 512.

Le résultat d'une conversion de type peut être indéterminé : par exemple, lors des conversions

```
long double ⟶ double, float, long, short
double ⟶ float, long, short
float ⟶ long, short
```

si la valeur de l'expression à convertir est, en valeur absolue, trop grande pour être exprimée dans le type d'arrivée. Par contre, dans le cas de conversions entre entiers longs, entiers courts et caractères, si la valeur convertie dépasse la capacité d'expression du type d'arrivée, la valeur à convertir est tronquée et seuls les bits de poids faible sont conservés. Les conversions de pointeurs seront développées dans la section 6.1.

3.3.3 Expressions arithmétiques et logiques

Expressions arithmétiques

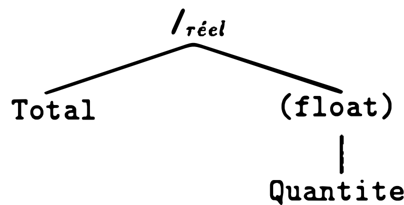
Ce sont les expressions les plus classiques des langages de programmation. La syntaxe des opérateurs arithmétiques est la suivante :

expression-additive-unaire	:	`+` *expression*
		`-` *expression*
expression-additive	:	*expression* `+` *expression*
		expression `-` *expression*
expression-multiplicative	:	*expression* `*` *expression*
		expression `/` *expression*
		expression `%` *expression*

La sémantique de ces opérateurs est résumée dans le tableau 3.4.

Les opérateurs `+`, `-`, `*` et `/` peuvent être, selon le type de leurs opérandes, des opérateurs arithmétiques entiers ou réels. Par exemple, `3/2` vaut 1 et `3.0/2.0` vaut 1.5. L'expression $\langle e_1 \rangle$`/`$\langle e_2 \rangle$ n'est pas définie si la valeur de $\langle e_2 \rangle$ est nulle. L'opérateur *modulo* `%` calcule le reste de la division entière; son second opérande doit également être non nul.

Lorsque les types des deux opérandes sont différents, il y a conversion implicite dans le type le plus fort. Par exemple, si `Total` est un réel et `Quantite` un entier, l'expression `Total / Quantite` est traduite par le compilateur en :



Les deux opérandes de cette nouvelle expression étant de type réel, l'opérateur `/` est interprété comme une division réelle.

Comme nous le verrons dans la section 6.1, il est également possible d'utiliser des variables de type pointeur dans certaines expressions arithmétiques.

Expressions logiques

Les expressions logiques sont les suivantes :

expression-relation : *expression* `>` *expression*
expression `>=` *expression*
expression `<` *expression*
expression `<=` *expression*
expression `==` *expression*
expression `!=` *expression*
expression-négation: `!` *expression*
expression-et : *expression* `&&` *expression*
expression-ou : *expression* `||` *expression*

La sémantique de ces opérateurs est résumée dans le tableau 3.4. Comme dans le cas des expressions arithmétiques, il y a conversion implicite si les arguments sont des quantités numériques de types différents. Il est possible, sous certaines conditions, d'effectuer des comparaisons entre deux pointeurs, comme nous le verrons dans la section 6.1. Le résultat de l'expression

$$\langle e_1 \rangle \&\& \langle e_2 \rangle$$

est 1 si les valeurs e_1 et e_2 sont toutes deux non nulles et 0 sinon (on rappelle que 0 code la valeur booléenne *faux* et que tout autre entier est interprété comme la valeur *vrai*). Le résultat de

$$\langle e_1 \rangle || \langle e_2 \rangle$$

est 1 si l'une des deux valeurs e_1 et e_2 est non nulle et 0 sinon. Lors de l'évaluation de $\langle e_1 \rangle$ `&&` $\langle e_2 \rangle$, la sous-expression $\langle e_2 \rangle$ n'est évaluée que si la valeur

e_1 est *vrai*, c'est-à-dire est différente de 0. En effet, la valeur de *faux et* e_2 est toujours *faux*, quelle que soit la valeur e_2. Par conséquent, l'expression

```
i>=0 && i<sizeof v && v[i]!=0
```

est toujours sémantiquement correcte. En effet, l'expression `v[i]!=0`, et donc `v[i]` ne sera évaluée que si on a

$$0 \leq i < \textit{taille de } \texttt{v}$$

De même, lors de l'évaluation de $\langle e_1 \rangle$ `||` $\langle e_2 \rangle$, l'expression $\langle e_2 \rangle$ n'est évaluée que si la valeur e_1 est *faux*, c'est-à-dire nulle.

Opérateurs arithmétiques				
Syntaxe de l'expression $\langle e \rangle$	*Type des expressions* $\langle e_1 \rangle$	$\langle e_2 \rangle$	$\langle e \rangle$	*Valeur e de l'expression* $\langle e \rangle$
`+` $\langle e_1 \rangle$	`int`	–	`int`	e_1
	`float`	–	`float`	e_1
`-` $\langle e_1 \rangle$	`int`	–	`int`	$-_{int}\, e_1$
	`float`	–	`float`	$-_{float}\, e_1$
$\langle e_1 \rangle$ `+` $\langle e_2 \rangle$	`int`	`int`	`int`	$e_1 +_{int} e_2$
	`float`	`float`	`float`	$e_1 +_{float} e_2$
	t *	`int`	t*	$e_1 +_{int} e_2 * \mathrm{sizeof}(t)$
	`int`	t *	t*	$e_1 * \mathrm{sizeof}(t) +_{int} e_2$
$\langle e_1 \rangle$ `-` $\langle e_2 \rangle$	`int`	`int`	`int`	$e_1 -_{int} e_2$
	`float`	`float`	`float`	$e_1 -_{float} e_2$
	t *	`int`	t*	$e_1 -_{int} e_2 *_{int} \mathrm{sizeof}(t)$
	t *	t *	`int`	$(e_1 -_{int} e_2)/_{int}\mathrm{sizeof}(t)$
$\langle e_1 \rangle$ `*` $\langle e_2 \rangle$	`int`	`int`	`int`	$e_1 *_{int} e_2$
	`float`	`float`	`float`	$e_1 *_{float} e_2$
$\langle e_1 \rangle$ `/` $\langle e_2 \rangle$	`int`	`int`	`int`	$e_1 /_{int} e_2$
	`float`	`float`	`float`	$e_1 /_{float} e_2$
$\langle e_1 \rangle$ `%` $\langle e_2 \rangle$	`int`	`int`	`int`	e_1 *mod* e_2
Opérateurs logiques				
Syntaxe de l'expression $\langle e \rangle$	*Type des expressions* $\langle e_1 \rangle$	$\langle e_2 \rangle$	$\langle e \rangle$	*Valeur e de l'expression* $\langle e \rangle$
$\langle e_1 \rangle$ `<` $\langle e_2 \rangle$			{0,1}	$e_1 < e_2$
$\langle e_1 \rangle$ `<=` $\langle e_2 \rangle$	entiers, réels		{0,1}	$e_1 \leq e_2$
$\langle e_1 \rangle$ `>` $\langle e_2 \rangle$	ou pointeurs		{0,1}	$e_1 > e_2$
$\langle e_1 \rangle$ `>=` $\langle e_2 \rangle$	(les 2 opérandes		{0,1}	$e_1 \geq e_2$
$\langle e_1 \rangle$ `==` $\langle e_2 \rangle$	de même type)		{0,1}	$e_1 = e_2$
$\langle e_1 \rangle$ `!=` $\langle e_2 \rangle$			{0,1}	$e_1 \neq e_2$
`!` $\langle e_1 \rangle$	`int`	–	{0,1}	*non* e_1
$\langle e_1 \rangle$ `&&` $\langle e_2 \rangle$	`int`	`int`	{0,1}	e_1 *et* e_2
$\langle e_1 \rangle$ `||` $\langle e_2 \rangle$	`int`	`int`	{0,1}	e_1 *ou* e_2

Tableau 3.4: Expressions arithmétiques et logiques

A propos du type booléen

Le type booléen n'est pas explicitement défini dans le langage C. Le résultat d'une expression logique est converti en entier lors de sa réutilisation dans une expression. Réciproquement, lorsqu'un entier doit être converti en booléen, la valeur nulle est convertie en *faux*, et toute valeur différente de zéro est convertie en *vrai*. En résumé, on a

$$\begin{array}{ccccc} \mathbb{Z}^* & \longrightarrow & \textit{vrai} & \longrightarrow & 1 \\ 0 & \longrightarrow & \textit{faux} & \longrightarrow & 0 \end{array}$$

Une conséquence est la possibilité d'utiliser une expression arithmétique dans des circonstances où on attend une expression logique (test dans une structure de contrôle, premier argument d'une expression conditionnelle...). Par exemple, l'instruction

```
if (fin - indice)
    indice++ ;
```

est équivalente à

```
if (fin - indice != 0)
    indice++ ;
```

Cependant, pour des raisons de lisibilité on évitera la première forme.

Dans certains cas, on s'autorise cependant l'utilisation d'expressions arithmétiques à la place d'expressions logiques. Les cas les plus fréquents sont le parcours d'une chaîne de caractère ou celui d'une liste chaînée. Par exemple, dans la fonction suivante, le test de la boucle *tant que* est effectué sur une expression de type caractère.

```
int chaine_longueur(char *chaine)
{
    int longueur = 0;

    while (*chaine++)
        longueur++;
    return longueur;
}
```

Cette fonction calcule la longueur de la chaîne de caractères qu'elle reçoit en paramètre. En effet, une chaîne de caractères correcte est terminée par le caractère `\0`, de code ASCII 0. La conversion de cette valeur en booléen donnant *faux*, l'itération s'effectue pour tous les caractères de la chaîne jusqu'au caractère `\0`. Cette écriture, qui peut sembler audacieuse, fait cependant partie de la culture de tout programmeur C.

3.3.4 Arithmétique booléenne et manipulation de bits

Opérateurs arithmétiques "bit à bit"

Les quatre opérateurs arithmétiques *bit à bit* correspondent aux quatre opérations classiques de l'algèbre booléenne : le *non*, le *et*, le *ou* et le *ou exclusif*. Leur syntaxe est la suivante :

expression-booléenne : `~` *expression*
expression `&` *expression*
expression `|` *expression*
expression `^` *expression*

La sémantique de ces opérateurs est résumée dans le tableau 3.5.

Une première opération classique effectuée avec ces opérateurs est le test de la valeur d'un bit dans un mot. Par exemple, l'expression `etat & 0x04` est différente de zéro si le troisième bit à partir de la droite de `etat` est à 1; elle est nulle sinon. En effet, si on considère pour simplifier des mots écrits sur huit bits, l'écriture en base 2 de la constante `0x04` est `00000100`, et celle de `etat` est $b_7b_6b_5b_4b_3b_2b_1b_0$ avec $b_i = 0$ ou $b_i = 1$. Sachant que d'une part

$$b_i \;\&\; 0 = 0$$
$$b_i \;\&\; 1 = b_i$$

et que d'autre part, le *et* est effectué *"bit à bit"*, le résultat de `etat & 0x04`, c'est-à-dire de

	0	0	0	0	0	1	0	0
&	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0

est `00000`b_2`00`, c'est-à-dire 0 si b_2 est nul, et 4 sinon.

Une autre opération très classique est le changement de la valeur d'un bit dans un mot. Par exemple, dans l'expression

```
etat | 0x04
```

le bit b_2 du résultat vaut 1, et les autres bits restent inchangés. De même, la mise à zéro du bit b_2 s'écrit

```
etat & ~0x04
```

où `~0x04` est le complément *"bit à bit"* de `0x04`. On a forcé un bit à un dans le premier cas, et à zéro dans le second.

On peut également masquer une partie d'un mot binaire. Par exemple, l'octet de poids faible d'une valeur entière `val` s'obtient par `val & 0xff`, le test *"⟨val⟩ est un entier pair"* peut s'écrire ⟨*val*⟩ `& 1`... Ces constructions peuvent être parfois difficiles à lire. Il est intéressant, dans ce cas, de les exprimer sous la forme de pseudo-fonctions du préprocesseur (voir section 5.1) :

```
#define OCTET_MINEUR(n)    ((n) & 0xff)
#define EST_PAIR(n)        ((n) & 0x1)
```

Opérateurs de décalage

Les deux opérateurs de décalage sont le décalage à droite et le décalage à gauche. Leur syntaxe est :

expression-décalage : *expression* `>>` *expression*
expression `<<` *expression*

La valeur de l'expression

$$\langle e_1 \rangle \texttt{ << } \langle e_2 \rangle$$

est obtenue en décalant de e_2 positions vers la gauche tous les bits de l'écriture binaire de e_1, et en complétant à droite par des zéros. Par exemple, si **val** est un entier court, dont l'écriture binaire est $b_{15}b_{14}\ldots b_0$, la valeur de **val<<1** s'écrit en binaire $b_{14}b_{13}\ldots b_0 0$. On notera que, lorsqu'il n'y a pas dépassement de capacité, `val << n` vaut $val \times 2^n$.

Le **décalage à droite** est dit *logique* lorsqu'on complète à gauche par des zéros, et *arithmétique* lorsqu'on recopie le bit le plus à gauche. Par exemple, un décalage de trois positions binaires à droite de l'octet 10001101 vaut 00010001 s'il est logique, et 11110001 s'il est arithmétique. On dit dans ce cas qu'il y a **extension du bit de signe**.

En C, un décalage à droite est logique si l'opérande de gauche est une quantité non signée (type `unsigned`). Dans le cas contraire, le résultat peut dépendre du compilateur; en général cependant, si l'opérande de gauche est une quantité signée, le décalage est arithmétique. On notera donc que si $val > 0$, `val >> n` calcule le résultat de la division entière $\frac{val}{2^n}$.

Arithmétique booléenne				
Syntaxe de l'expression ⟨*e*⟩	*Type des expressions* ⟨e_1⟩	⟨e_2⟩	⟨*e*⟩	Valeur e de *l'expression* ⟨*e*⟩
`~` ⟨e_1⟩	int	–	int	complément de e_1
⟨e_1⟩ `&` ⟨e_2⟩	int	int	int	e_1 *et-logique* e_2
⟨e_1⟩ `|` ⟨e_2⟩	int	int	int	e_1 *ou-inclusif* e_2
⟨e_1⟩ `^` ⟨e_2⟩	int	int	int	e_1 *ou-exclusif* e_2
Décalages				
Syntaxe de l'expression ⟨*e*⟩	*Type des expressions* ⟨e_1⟩	⟨e_2⟩	⟨*e*⟩	*Valeur e de l'expression* ⟨*e*⟩
⟨e_1⟩ `>>` ⟨e_2⟩	int	int	int	décalage arithmétique de e_1, de e_2 bits vers la droite.
	unsigned	int	unsigned	décalage logique de e_1, de e_2 bits vers la droite.
⟨e_1⟩ `<<` ⟨e_2⟩	int	int	int	décalage de e_1, de e_2 bits vers la gauche.

Tableau 3.5: Expressions de manipulation de bits

On utilise conjointement les masquages et les décalages pour extraire ou modifier des sous-mots d'un mot. Par exemple, l'octet de poids fort d'un entier court `etat` s'obtient par

```
(etat >> 8) & 0xff
```

La pseudo-fonction `OCTET_MAJEUR` calculant l'octet de poids fort d'un entier court s'écrit par conséquent :

```
#define OCTET_MAJEUR(n) (((n) >> 8) & 0xff)
```

Voici pour terminer cette section l'exemple d'une fonction comptant le nombre de bits à un dans un mot.

```
compte_bits.c
#include "compte_bits.h"

int compte_bits(unsigned long mot)
{
    int nombre = 0;

    for (;;)
    {
        if (mot == 0)
            return nombre;
        if (mot & 0x1 == 1)
            nombre++;
        mot = mot >> 1;
    }
}

compte_bits.c
```

3.3.5 Manipulation d'adresses

Une adresse est la référence en mémoire à une variable ou à une fonction. Les deux opérateurs permettant de manipuler des adresses sont `*` et `&`; leur syntaxe est la suivante :

expression-indirection : `*` *expression*
expression-référence : `&` *expression*

L'opérateur `&` permet d'obtenir l'adresse mémoire d'une variable. Par exemple, la valeur de l'expression `&pas` est l'adresse de la variable `pas`. Il peut s'appliquer sur toute expression qui est une *g-valeur*, puisqu'une *g-valeur* est une référence à un emplacement mémoire. Par exemple, compte tenu des déclarations suivantes :

```
struct enregistrement
{
    char nom[TNOM];
    long reference;
};
struct enregistrement enr;
```

on peut former les expressions :

```
&enr
&enr.nom[0]
&enr.nom[TNOM-1]
&enr.reference
```

qui calculent respectivement l'adresse :
- de la structure `enr`;
- du premier caractère du champ `nom`;

– du dernier caractère du champ nom;
– du champ reference.

On peut également utiliser la pseudo-fonction offsetof, définie dans le fichier en-tête <stddef.h>, pour connaître le décalage entre un champ d'une structure et le début de cette structure. Par exemple, le décalage du champ reference est donné par l'expression

```
offsetof(struct enregistrement, reference)
```

L'exécution du programme suivant implementation_de_enr suivant :

```
═══ implementation_de_enr.c ═══
#include <stdio.h>
#include <stddef.h>

#define TNOM 14

struct enregistrement
{
    char nom[TNOM];
    long reference;
};
struct enregistrement enr;

main()
{
    printf("Adresse de\n");
    printf("  enr: %p\n", &enr);
    printf("  enr.nom[0]: %p\n", &enr.nom[0]);
    printf("  enr.nom[TNOM-1]: %p\n", &enr.nom[TNOM-1]);
    printf("  enr.reference: %p\n", &enr.reference);
    printf("Decalage du champ reference: %d\n",
           offsetof(struct enregistrement, reference));
}
═══ implementation_de_enr.c ═══
```

affiche l'implantation mémoire de la structure enr :

```
$ implementation_de_enr
Adresse de
  enr: 40c0
  enr.nom[0]: 40c0
  enr.nom[TNOM-1]: 40cd
  enr.reference: 40d0
Decalage du champ reference: 16
$
```

On rappelle que sous UNIX, l'éditeur de liens rajoute le caractère _ devant chaque symbole (remarque 5, page 32). On notera le réalignement opéré par le compilateur pour faire commencer le champ reference sur une frontière de double mot.

Remarque 15 *Si ⟨vect⟩ est un identificateur de vecteur, les expressions ⟨vect⟩ et* &⟨vect⟩[0] *sont équivalentes. L'expression* &⟨vect⟩ *est en général acceptée par les compilateurs comme équivalente aux deux formes précédentes.*

Une adresse peut être stockée dans une variable de type pointeur. Si p_enr est déclaré de type struct enregistrement *, l'expression p_enr = &enr fait pointer p_enr sur le début de la structure. L'opérateur d'**indirection** * retourne la valeur de *"l'objet pointé"*. Dans l'exemple précédent, il est équivalent d'écrire enr et (*p_enr). Rappelons enfin que l'expression (*p_enr).nom[0], qui renvoie la valeur du premier caractère du champ nom, peut être abrégée en p_enr->nom[0].

Nous reviendrons plus longuement sur les manipulations d'adresses et de pointeurs dans la section 6.1. Le tableau 3.6 donne un résumé de la syntaxe et de la sémantique de ces opérateurs.

Syntaxe de l'expression $\langle e \rangle$	*Type des expressions* $\langle e_1 \rangle$	$\langle e \rangle$	*Valeur e de l'expression* $\langle e \rangle$
& $\langle e_1 \rangle$	t *(g-valeur)*	t *	*adresse de l'objet de type t référencé par* $\langle e_1 \rangle$
* $\langle e_1 \rangle$	t *	t	*valeur de l'objet de type t rangé à l'adresse mémoire* $\langle e_1 \rangle$

Tableau 3.6: Expressions de manipulation d'adresse

3.3.6 Taille d'un objet

La taille en octets d'un objet peut être calculée au moyen de l'opérateur sizeof :

expression-tailllede : sizeof *expression*
sizeof (*expression-de-type*)

La taille d'une expression est la taille d'une variable de même type que cette expression. Par exemple, sizeof 1.0 et sizeof (float) ont la même valeur qui est le nombre d'octets nécessaires pour coder un réel. Le résultat est une constante sans signe (voir tableau 3.7).

Syntaxe de l'expression $\langle e \rangle$	*Type des expressions* $\langle e_1 \rangle$	$\langle e \rangle$	*Valeur e de l'expression* $\langle e \rangle$
sizeof $\langle e_1 \rangle$	*- quelconque -*	unsigned	*taille en octets d'un objet du type de* $\langle e_1 \rangle$
sizeof ($\langle t \rangle$)	*expression de type*	unsigned	*taille en octets d'un objet de type* $\langle t \rangle$

Tableau 3.7: Expression *taille-de*

Cet opérateur est, entre autres, très utile lors de la mise en œuvre des entrées-sorties de bas niveau (fonctions read et write) et de l'allocation dynamique de mémoire (fonction malloc); ces fonctionnalités seront présentées

plus loin. Un autre exemple d'utilisation de l'opérateur `sizeof` est le calcul du nombre d'éléments d'un vecteur dont la taille est fixée par une initialisation. Considérons par exemple la définition de menu déroulant présentée page 81. Le nombre d'éléments du vecteur est donné par l'expression

```
(sizeof menu / sizeof menu[0])
```

qu'on peut associer à une pseudo-constante du préprocesseur :

```
#define NOMBRE_DE_CHOIX   (sizeof menu / sizeof menu[0])
```

On aurait également pu écrire

```
#define NOMBRE_DE_CHOIX   (sizeof menu / sizeof(struct choix))
```

Il est de cette façon possible d'ajouter un choix dans le menu sans avoir à redéfinir la dimension du vecteur, ce qui élimine des causes d'erreurs.

La généralisation de cette construction peut être effectuée au moyen de la pseudo-fonction suivante :

```
#define NOMBRE_D_ELEMENTS(vect) (sizeof(vect)/sizeof(vect)[0])
```

Le programme suivant illustre l'utilisation de cette construction :

```
======= nombre_d_elements.c =======
#include <stdio.h>

#define NOMBRE_D_ELEMENTS(vect) (sizeof(vect)/sizeof(vect)[0])

main()
{
   int v1[10];
   double v2[55];

   printf("v1 contient %d elements, et v2 en contient %d.\n",
          NOMBRE_D_ELEMENTS(v1),
          NOMBRE_D_ELEMENTS(v2));
}
======= nombre_d_elements.c =======
```

Son exécution se déroule de la manière suivante :

```
$ nombre_d_elements
v1 contient 10 elements, et v2 en contient 55.
$
```

Remarque 16 *L'opérateur* `sizeof` *appliqué sur une constante chaîne de caractères retourne sa taille réelle, incluant le caractère* `NULL` *de fin de chaîne. Par exemple, l'expression* `sizeof "cinq"` *vaut 5.*

3.3.7 Listes d'expressions et expressions conditionnelles

Ces formes d'expressions permettent de regrouper des sous-expressions afin de les évaluer en séquence dans le cas de la liste, ou selon le résultat d'une

sous-expression dans le cas de l'expression conditionnelle (voir tableau 3.8).

expression-liste : *expression* , *expression*
expression-conditionnelle : *expression* ? *expression* : *expression*

La valeur d'une liste d'expressions est celle de la dernière de la liste. L'évaluation d'une expression $\langle e \rangle$ de la forme

$$\langle e_1 \rangle \texttt{?} \langle e_2 \rangle \texttt{:} \langle e_3 \rangle$$

dépend de la valeur e_1, interprétée comme un booléen. Si e_1 est *vrai*, c'est-à-dire non nulle, la valeur de $\langle e \rangle$ est égale à e_2, et elle est égale à e_3 sinon. Les listes sont utiles pour enchaîner plusieurs expressions à effet de bord, comme des affectations par exemple, dans une situation où il n'est pas possible d'utiliser d'instruction. C'est le cas avec la structure de contrôle `for` dont la syntaxe est :

$$\texttt{for (}\langle e_1 \rangle\texttt{; } \langle e_2 \rangle\texttt{; } \langle e_3 \rangle\texttt{)}$$
$$\ldots$$

les arguments $\langle e_1 \rangle$, $\langle e_2 \rangle$ et $\langle e_3 \rangle$ étant des expressions. Les expressions listes permettent d'effectuer plusieurs effets de bord dans chacune des expressions de contrôle de la boucle :

```
for (total=0, p=debut; p!=NULL; total++, p=p->suivant)
    ...
```

Syntaxe de l'expression $\langle e \rangle$	*Type des expressions* $\langle e_1 \rangle$	$\langle e_2 \rangle$	$\langle e_3 \rangle$	$\langle e \rangle$	*Valeur e de l'expr.* $\langle e \rangle$
$\langle e_1 \rangle \texttt{,} \langle e_2 \rangle$	t_1	t_2	–	t_2	e_2
$\langle e_1 \rangle \texttt{?} \langle e_2 \rangle \texttt{:} \langle e_3 \rangle$	t_1	t_2	t_3	$max(t_2, t_3)$	*si* $e_1 \neq 0$ *alors* e_2 *sinon* e_3

Tableau 3.8: Liste d'expressions et expression conditionnelle

L'expression conditionnelle est très utile lors de la définition de pseudo-fonctions :

```
#define MAXIMUM(a,b) ((a)>(b) ? (a) : (b))
#define DEPILER()    (pile_vide() ? INDEFINI : *ppile--)
#define PUTCHAR(c)   (printf(((c)<' ') ? "\%03d" : "%c", (c));
```

Dans d'autres cas, elle remplace avantageusement une instruction de test *si-sinon* qui alourdirait l'écriture :

```
printf("Il est %d heure%s\n", heure, heure>1 ? "s" : "");
```

Cette dernière utilisation est d'ailleurs suffisamment fréquente pour justifier, elle aussi, une définition.

```
======== pluriel.h ========
#ifndef PLURIEL_H
#define PLURIEL_H
```

```
#define PLURIEL(n) ((n)>1 ? "s" : "")

#endif  /* PLURIEL_H */
```
pluriel.h

Remarque 17 *Le type de* $\langle e\rangle ? \langle e_1\rangle : \langle e_2\rangle$ *dépend des types respectifs de* $\langle e_1\rangle$ *et de* $\langle e_2\rangle$. *Par conséquent, ces deux sous-expressions doivent être de même type, ou de deux types comparables; dans ce second cas, le type de* $\langle e\rangle$ *est le plus grand des deux types de* $\langle e_1\rangle$ *et* $\langle e_2\rangle$.

3.3.8 Opérateurs à effet de bord

Les opérateurs les plus originaux du langage C sont les opérateurs à effet de bord. Un **effet de bord** sur une expression peut être défini comme une opération qui modifie son contexte d'évaluation. Ce contexte est l'ensemble des valeurs courantes des variables intervenant dans son évaluation. Les effets de bord les plus courants sont la modification du contenu d'une variable et les opérations d'entrée-sortie. Les opérateurs à effet de bord du langage C sont de la première catégorie : affectation, incrémentation, etc.

Auto-incrémentations et auto-décrémentations

Ces opérateurs permettent d'ajouter ou d'enlever 1 au contenu d'une variable. Il existe deux opérateurs préfixes, et deux postfixes; les deux premiers sont la *pré-incrémentation* et la *pré-décrémentation*, et les deux derniers, la *post-incrémentation* et la *post-décrémentation*. Leur syntaxe est :

expression-incrémentation : ++ *expression*
expression ++
expression-décrémentation : -- *expression*
expression --

Ces opérateurs ne s'appliquent que sur des *g-valeurs*. La valeur d'une telle expression est l'ancienne valeur de l'opérande avec les opérateurs postfixes, et la nouvelle (c'est-à-dire l'ancienne incrémentée ou décrémentée de 1) avec les opérateurs préfixes. Considérons par exemple la suite d'expressions

```
x = 2
y = x++
z = ++x
```

que l'on supposera être évaluées dans cet ordre. Après l'évaluation de la seconde, `y` vaut 2, c'est-à-dire la valeur de `x` avant l'incrémentation, et `x` vaut 3. Après l'évaluation de la troisième, `z` et `x` valent toutes les deux 4.

Ces opérateurs sont très utilisés, pour gérer des indices de boucle, des pointeurs, des indices de piles... Par exemple, si `pile` est un vecteur et `sommet` l'indice du sommet de pile, l'expression

```
pile[++sommet] = valeur
```

empile `valeur` et l'expression

```
valeur = pile[sommet--]
```

dépile un élément en l'affectant à la variable `valeur`. Itérer une action *nb* fois s'écrit

```
while (nb-- > 0)
    ...
```

Si `p` est un pointeur positionné sur le début d'une chaîne de caractères non vide, l'instruction

```
while (*++p)
    ;
```

positionne `p` sur le marqueur de fin de chaîne.

Affectation simple

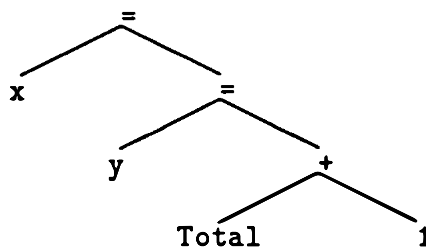
L'affectation est la construction de base des langages de programmation impératifs. En C, l'affectation est une expression, au même titre que l'addition ou la comparaison. Elle s'écrit au moyen de l'opérateur = dont la syntaxe est :

expression-affectation : expression = *expression*

Sa valeur et son type sont la valeur et le type de la sous-expression de droite. Au moment de son évaluation, cette expression génère un effet de bord : la recopie de sa valeur dans la variable référencée par la sous-expression de gauche. Cette dernière doit, par définition, être une *g-valeur* [7]. Considérons par exemple, l'expression

```
x = y = Total + 1
```

Son arbre de syntaxe est



Si la valeur de `Total` est 3, la valeur de `Total+1` est 4; celle des sous-expressions `y=(Total+1)` et `x=(y=Total+1)` est donc également 4. Parallèlement à cette évaluation est effectué un effet de bord sur les variables `x` et `y` qui reçoivent la valeur 4.

L'implémentation de l'affectation sous la forme d'une expression et non d'une instruction peut surprendre. C'est d'ailleurs une source d'erreur classique chez les débutants en C. Cette erreur est illustrée par le programme `nultest` suivant, qui bien que parfaitement correct du point de vue syntaxique, produit un résultat inattendu :

[7]Rappelons que le terme *g-valeur* signifie : valeur pouvant être utilisée comme membre gauche d'une affectation.

```
═══════ nultest.c ═══════════════════════════
#include <stdio.h>

main()
{
    int x = 0;

    printf("Valeur de x : %d\n", x);
    if (x=0)
        printf(" x est nul\n");
    else
        printf(" x n'est pas nul\n");
}
═══════════════════════════ nultest.c ═══════
```

La compilation de `nultest.c` se déroule sans problème, mais son exécution réserve une surprise.

```
$ gcc -g nultest.c -o nultest
$
$ nultest
Valeur de x : 0
x n'est pas nul
$
```

L'erreur est bien sûr contenue dans le test : on a écrit **x=0** au lieu de **x==0**; Les deux écritures

```
if (x = 0)
    ...
```

et

```
if (x == 0)
    ...
```

sont syntaxiquement correctes. Dans la première, le résultat de **x=0** est la valeur du membre droit de l'expression, c'est-à-dire zéro, qui est converti en *Faux*, ce qui explique le comportement au premier abord curieux du programme `nultest`.

On remarquera que dans le *langage algorithmique libre*, on utilise couramment l'affectation pour former des expressions. On dit par exemple :

> **tant-que** *le caractère suivant* c *n'est pas* EOF
> **faire**
> ...

Cela se traduit directement en C par :

```
while ((c = getchar()) != EOF)
    ...
```

Enfin, la remarque effectuée au sujet des opérateurs de condition et de liste est également valable : il est possible d'effectuer des affectations lorsque la syntaxe interdit l'utilisation d'une instruction.

Affectations combinées

Il s'agit d'un ensemble d'opérateurs combinant un opérateur binaire de l'arithmétique classique ou de l'arithmétique booléenne avec une affectation. Ces opérateurs sont au nombre de dix :

affectation-combinée : `+=` *expression*
`-=` *expression*
`*=` *expression*
`/=` *expression*
`%=` *expression*
`&=` *expression*
`|=` *expression*
`^=` *expression*
`<<=` *expression*
`>>=` *expression*

De façon générale, une expression de la forme

$$\langle e_1 \rangle \ op\texttt{=} \ \langle e_2 \rangle$$

est équivalente à

$$\langle e_1 \rangle \ \texttt{=} \ \langle e_1 \rangle \ op \ \langle e_2 \rangle$$

Par exemple, la valeur de `x += 3` est $x + 3$, qui devient la nouvelle valeur de `x` après l'évaluation.

Ces opérateurs sont en fait très naturels; en effet, ainsi que le font remarquer Kernighan et Ritchie, ≪*On dit "ajouter 2 à i", ou encore "incrémenter i de 2", et non pas "prendre i, ajouter 2, puis ranger le résultat dans i"*≫. Voici quelques exemples classiques d'expressions utilisant des affectations combinées :

`valeur >>= i`	:	décaler `valeur` de `i` bits à droite;
`caractere &= 0x7f`	:	masquer le bit de poids fort de `caractere`;
`Marque |= 0x1<<(n-1)`	:	mettre à un le $n^{\text{ème}}$ bit de `Marque`.

Affectation et vecteurs

On a vu qu'un identificateur de vecteur n'est pas une g-valeur. Il n'est donc pas possible d'effectuer une recopie de vecteur au moyen d'une affectation. Par contre, une structure est une g-valeur et peut apparaître en membre gauche d'une affectation. S'il s'avère utile d'effectuer au moyen d'une affectation la recopie de tout un vecteur, il suffit donc de l'englober dans une structure.

L'exemple suivant montre la déclaration d'un vecteur de `NBJOURS` réels encapsulée dans une structure de nom `struct total`.

```
affect_vecteur.c
#define NBJOURS 7

#define ELEM(x,i)    ((x)._[i])

/* Au lieu de : float total[NBJOURS]; */
struct total {
    float _[NBJOURS];
```

```
};

main()
{
    struct total tmp, res;
    int i;

    for (i=0; i<NBJOURS; i++)
        ELEM(tmp, i) = 1.0 / (float)(i+1);

    res = tmp;

    for (i=0; i<NBJOURS; i++)
        printf("%4.2f ", ELEM(res, i));
    printf("\n");
}
```
affect_vecteur.c

L'exemple utilise les variables `tmp` et `res`. La première est initialisée au moyen d'une boucle et la seconde par une simple affectation.

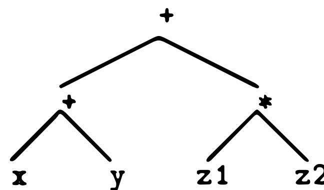
```
$ affect_vecteur
1.00 0.50 0.33 0.25 0.20 0.17 0.14
$
```

3.4 A propos des effets de bord

Nous avons présenté dans la section précédente les différents opérateurs d'expressions produisant des effets de bord. Lorsqu'une expression est construite sans opérateur à effet de bord, le résultat de son évaluation est indépendant de l'ordre selon lequel sont évaluées ses sous-expressions. Considérons par exemple l'expression

`x + y + z1*z2`

Les différentes règles de priorité et d'associativité permettent de déterminer sans ambiguïté son arbre de syntaxe.



Mais ces différentes règles (priorité de `*` par rapport à `+` et associativité gauche-droite pour `+`) ne permettent pas de déterminer laquelle des deux sous-expressions, de `x+y` et `z1*z2`, est évaluée la première.

En pratique, les compilateurs peuvent, pour des raisons d'optimisation de code, privilégier un ordre d'évaluation particulier selon la structure de l'expres-

sion. Sur cet exemple, l'ordre d'évaluation n'a aucune incidence sur le résultat, puisqu'aucun opérateur ne modifie le contexte d'évaluation de l'expression.

Par contre, l'utilisation d'opérateurs à effet de bord peut produire des expressions ambiguës, dont la valeur dépend du compilateur. Considérons l'expression

```
--x + y++
```

et supposons qu'avant son évaluation, les variables `x` et `y` contiennent respectivement les valeurs 3 et 7. L'évaluation est effectuée de la façon suivante :

- l'expression `y` vaut 7;
- l'expression `y++` vaut également 7; la variable `y` est incrémentée de 1 et contient après évaluation la valeur 8;
- l'expression `x` vaut 3;
- l'expression `--x` vaut 2 car la variable `x` est décrémentée avant son utilisation;
- l'expression `--x + y++` vaut donc 9.

De plus, les variables `x` et `y` ont été modifiées, et contiennent après l'évaluation, respectivement 2 et 8. Ce premier exemple ne comporte aucune ambiguïté.

Considérons maintenant l'expression

```
--x + x++
```

Cette expression est ambiguë; en effet, sa valeur diffère selon que son évaluation commence par `--x` ou par `x++`.

Les différents compilateurs que nous avons testés utilisaient tous le même schéma de traduction :

1) génération du code réalisant les pré-incrémentations et les pré-décrémentations;
2) génération du code évaluant l'expression;
3) génération du code réalisant les post-incrémentations et les post-décrémentations.

Ce schéma convient parfaitement pour traduire

```
a = --x + y++
```

On obtient une suite d'instructions élémentaires de la forme :

```
DECREMENTER   x
ADDITIONNER   x ET y DANS R
RANGER        R DANS a
INCREMENTER   y
```

qui produira bien le résultat attendu. Par contre, pour l'expression

```
a = --x + x++
```

on obtient la suite d'instructions

```
DECREMENTER   x
ADDITIONNER   x ET x DANS R
RANGER        R DANS a
INCREMENTER   x
```

qui produira à l'exécution un résultat à priori imprévisible. On retiendra de tout ceci la règle suivante :

Dans une expression, une variable qui subit un effet de bord doit être utilisée une seule fois, sauf éventuellement dans les opérandes d'un opérateur dont l'ordre d'évaluation est déterministe. Les opérateurs dont l'ordre d'évaluation est déterministe sont : l'opérateur d'expression conditionnelle (`? :`), l'opérateur de liste (`,`), et les connecteurs logiques (`&&` et `||`).

Cette règle s'applique également aux paramètres de fonctions . Par exemple, l'appel :

```
f(x, ++x)
```

est ambigu. En effet, l'ordre d'évaluation des paramètres effectifs dans un appel de fonction n'est pas fixé en C (ni d'ailleurs dans la plupart des autres langages de programmation). Le programme `effet_de_bord.c` suivant démontre ce danger.

```
effet_de_bord.c
main()
{
    int x;

    x = 666;
    printf("%d %d\n", x, x++);
}
effet_de_bord.c
```

Dans ce cas, on peut voir que le troisième paramètre `x++` a été évalué avant le second.

```
$ effet_de_bord
667 666
$
```

Enfin, un effet de bord peut également être provoqué par l'exécution d'une fonction. Attention, donc, à ne pas écrire d'instruction de la forme

```
empiler(eval(depiler(), depiler(), depiler()));
```

dont le comportement peut être différent de celui de la suite d'instructions :

```
op = depiler();
x1 = depiler();
x2 = depiler();
empiler(eval(op, x1, x2));
```

Chapitre 4

Les instructions

Les précédents chapitres ont montré comment former les objets de base (constantes, variables et fonctions), et comment les combiner pour former des expressions. Les instructions sont des constructions se plaçant au-dessus des expressions dans la hiérarchie des concepts C. Elles sont formées à partir d'instructions de base et de **structures de contrôle**. Contrairement aux expressions, les instructions n'ont ni type ni valeur. Lorsqu'on forme une instruction à partir d'une expression, la valeur de cette dernière est perdue.

On distingue trois familles d'instructions :

- les instructions élémentaires;
- les instructions composées, construites au moyen des structures de contrôles et d'instructions;
- les instructions d'échappement.

4.1 Instructions élémentaires

Une **instruction élémentaire** est formée d'une expression terminée par un point-virgule. Par exemple, les instructions

```
x++;
premier = 1;
max = (k1 > k2) ? k1 : k2 ;
```

sont des instructions élémentaires formées à partir des expressions

```
x++
premier = 1
max = (k1 > k2) ? k1 : k2
```

Plus précisément, la syntaxe d'une instruction élémentaire est :

instruction-élémentaire : expression ;

N'importe quelle expression peut former une instruction, même lorsqu'elle ne génère pas d'effet de bord. Une telle instruction est parfaitement inutile; il peut de plus s'agir d'une erreur dans la saisie du programme, erreur qui ne sera pas signalée par le compilateur puisque l'instruction est syntaxiquement correcte.

En fait, ce problème est de nature sémantique. Il n'est, en effet, pas toujours possible de déterminer syntaxiquement si une expression génère ou non un effet de bord. Par exemple, l'instruction

```
0;
```

représente un cas particulier pour lequel on sait décider syntaxiquement qu'elle est inutile; elle est à rapprocher de la construction

si (FAUX) alors
...

qui est syntaxiquement correcte dans tous les langages structurés, mais est sémantiquement sans intérêt.

La structure de bloc, délimitée par les symboles { et } permet de regrouper une liste d'instructions en une seule. Il est de plus possible de placer, en tête du bloc, des déclarations de variables locales à ce bloc. La syntaxe d'un bloc est :

bloc : {
  [*liste-de-déclarations*]
  [*liste-d'instructions*]
}

Nous avons déjà présenté le mécanisme de déclaration de variables dans un bloc en 2.6.1.

Voici deux exemples de regroupement d'instructions en bloc; le premier est le bloc principal d'une fonction

```
void iechange(int *x, int *y)
{
    int z = *x;

    x = *y;
    y = z;
}
```

et le second, la partie *alors* d'une instruction *si*

```
if (x > y)
{
    int z = x;

    x = y;
    y = z;
}
```

⊚ *Pour accroître la lisibilité d'un programme, on laissera une ligne vide entre la dernière ligne d'une liste de déclarations et la première d'une liste d'instructions.*

On notera que dans une liste d'instructions, le point-virgule fait partie de l'instruction qu'il termine, contrairement à d'autres langages où il est utilisé comme séparateur. La syntaxe d'une liste d'instructions est :

$$\begin{array}{c}\langle instruction\rangle\\ \langle instruction\rangle\\ \vdots\\ \langle instruction\rangle\end{array}$$

et non :

$$\begin{array}{c}\langle instruction\rangle\texttt{;}\\ \langle instruction\rangle\texttt{;}\\ \vdots\\ \langle instruction\rangle\end{array}$$

Par conséquent, lors de l'écriture d'une liste d'instructions élémentaires, on prendra garde à ne pas oublier le point-virgule de la dernière instruction, c'est-à-dire celui précédant l'accolade fermante.

4.2 Structures de contrôle

Les **structures de contrôle** du langage C sont très classiques : le test, diverses formes d'itérations, et l'aiguillage.

4.2.1 Test

La syntaxe du **test** est :

instruction-test : `if (`*expression*`)`
  partie-alors
 [`else`
  partie-sinon **]**

La *partie alors* et la *partie sinon* du test peuvent être indifféremment une instruction ou un bloc. Si la valeur de l'expression entre parenthèses est différente de zéro, la *partie alors* est exécutée. Sinon, dans le cas où le test comporte une *partie sinon*, celle-ci est exécutée. Considérons par exemple l'instruction

```
if (r.denominateur != 0)
    q = (float) r.numerateur / (float) r.denominateur;
else
{
    code_erreur = DIVISION_PAR_ZERO;
    erreur();
}
```

L'expression `r.denominateur` est d'abord évaluée. Si le résultat de cette évaluation est différent de zéro, l'instruction

```
q = (float) r.numerateur / (float) r.denominateur;
```

est exécutée, sinon ce sont les deux instructions du bloc

```
{
    code_erreur = DIVISION_PAR_ZERO;
    erreur();
}
```

qui le sont.

Remarque 18 *Bien que la forme* `if (r.denominateur)` *soit syntaxiquement et sémantiquement correcte, et équivalente à*

```
if (r.denominateur != 0)
```

elle est à éviter car elle suggère une mauvaise interprétation du type de la variable `r.denominateur` *lors de la lecture du programme. On réservera la forme* `if (`⟨*variable*⟩`)` *aux variables ne prenant que des valeurs booléennes, soit 0 ou 1.*

Lorsque plusieurs tests sont imbriqués, chaque *partie sinon* est reliée au *si* le plus proche qui n'est pas déjà associé à une *partie sinon*. Par exemple, l'instruction

```
if (x > 0)
    ecrire("positif");
else if (x < 0)
    ecrire("negatif");
else
    ecrire("nul");
```

est équivalente à

```
if (x > 0)
    ecrire("positif");
else
{
    if (x < 0)
        ecrire("negatif");
    else
        ecrire("nul");
}
```

Il est possible d'utiliser une instruction vide, réduite à un `;`, dans les parties *alors* et *sinon*. L'instruction

```
if (y != 0)
{
    if (x != 0)
        normaliser(&x, &y);
}
else
{
    code_erreur = DENOMINATEUR_NUL;
    erreur();
}
```

peut également s'écrire

```
if (y != 0)
    if (x != 0)
        normaliser(&x, &y);
    else
        ;
else
{
    code_erreur = DENOMINATEUR_NUL;
    erreur();
}
```

Cependant, pour des raisons de lisibilité, il est préférable d'expliciter les blocs au moyen d'accolades, comme dans l'exemple précédent.

4.2.2 Aiguillage

Une instruction d'**aiguillage** permet de tester la valeur d'une expression parmi un ensemble de constantes. La définition syntaxique que nous retiendrons pour l'aiguillage est la suivante :

```
instruction-aiguillage : switch(expression)
                         {
                           case expression-constante :
                           [ liste-d'instructions ]
                           ⋮
                         [ default:
                           [ liste-d'instructions ] ]
                         }
```

En réalité la syntaxe de l'aiguillage est plus générale. Par exemple, l'instruction

```
switch (c)
    case '\n':
        c = '\0';
```

est syntaxiquement correcte. Cependant, cette forme n'a que très peu d'intérêt, et nous nous en tiendrons, dans l'écriture de nos programmes, à la syntaxe restreinte donnée précédemment[1].

Toutes les expressions constantes figurant après un `case` doivent être différentes. Si la valeur de l'expression de contrôle est celle de la i[ème] expression constante c_i, l'exécution de l'aiguillage commence par la première instruction étiquetée par

`case` c_i`:`

Une liste d'instructions peut être vide. Dans le cas où l'étiquette `default` est présente et si l'expression n'est égale à aucune des constantes, l'exécution de l'aiguillage commence par la première instruction étiquetée par

`default:`

Il n'est pas obligatoire de placer l'étiquette `default` en dernière position.

[1] La forme la plus générale peut être utile lorsque C est utilisé comme langage intermédiaire par un autre compilateur ou un générateur de programme.

Par défaut, l'exécution se poursuit jusqu'à la dernière instruction du bloc; il est possible de forcer la terminaison de l'instruction d'aiguillage au moyen de l'instruction

```
break
```

Il est important de noter qu'en C l'aiguillage

```
switch(⟨exp⟩)
{
  case c1 :
    ⟨liste d'instructions1⟩
  case c2 :
    ...
  default:
    ⟨liste d'instructionsdef⟩
}
```

n'est pas équivalent à l'enchaînement de test

```
if (⟨exp⟩ == c1)
    ⟨liste d'instructions1⟩
else if (⟨expression⟩ == c2)
    ⋮
else
    ⟨liste d'instructionsdef⟩
```

puisque par défaut, l'exécution se poursuit jusqu'à la fin du bloc. Considérons l'exemple suivant :

sw.c

```
#include <stdio.h>

main(int argc, char *argv[])
{
    switch (argv[1][0])
    {
      case 'a' :
      case 'A' :
        printf("  - Passage en A\n");

      case 'b' :
      case 'B' :
        printf("  - Passage en B\n");

      default :
        printf("  - Instructions par defaut\n");
    }
}
```

sw.c

Le test de l'aiguillage porte sur le premier caractère du premier argument avec lequel est invoquée la commande, c'est-à-dire la valeur de l'expression `argv[1][0]`. Cette valeur détermine sur quel champ de l'aiguillage se poursuit l'exécution. Toutes les instructions entre le point d'entrée et la fin du bloc sont exécutées. Par exemple, si la commande est appelée avec la lettre `b` en

argument, les instructions exécutées seront :

```
printf(" - Passage en B\n");
printf(" - Instructions par defaut\n");
```

Ce programme peut produire les trois résultats suivants :

```
$ sw a
  - Passage en A
  - Passage en B
  - Instructions par defaut
$ sw B
  - Passage en B
  - Instructions par defaut
$ sw c
  - Instructions par defaut
$
```

Voici une seconde version du programme précédent utilisant l'instruction `break`; chaque champ forme, dans ce cas, un ensemble d'instructions disjoint des autres. Cette forme est équivalente à un enchaînement de tests.

```
======== sw_break.c ========
#include <stdio.h>

main(int argc, char *argv[])
{
    switch (argv[1][0])
    {
      case 'a' :
      case 'A' :
        printf("  - Passage en A\n");
        break;

      case 'b' :
      case 'B' :
        printf("  - Passage en B\n");
        break;

      default :
        printf("  - Instructions par defaut\n");
        break;
    }
}
======== sw_break.c ========
```

Dans cette nouvelle version du programme, les trois entrées de l'aiguillage sont des blocs de traitement disjoints :

```
$ sw_break A
  - Passage en A
$ sw_break b
  - Passage en B
$ sw_break c
```

```
  - Instructions par defaut
$
```

Le dernier `break` n'est pas nécessaire, mais il est préférable, au moins pour des raisons d'homogénéité, de le faire figurer explicitement.

Remarque 19 *Lorsqu'une liste d'instructions non vide n'est pas terminée par un* `break`, *on peut le signaler par un commentaire. On utilisera par exemple le commentaire* `/*NOBREAK*/` *reconnu par certains utilitaires.*

L'exemple suivant est une version simplifiée de la commande **WC(1)**, comptant le nombre de caractères, de mots et lignes lus sur l'entrée standard. On remarquera que le premier champ ne contient pas d'instruction `break`, le traitement des séparateurs de mots étant inclus dans celui des séparateurs de lignes.

```
========== compter.c ==========
#include <stdio.h>

#include "pluriel.h"

enum Etat
{
    Dans_un_mot,
    Hors_d_un_mot
};
typedef enum Etat Etat;

main()
{
    int  nombre_caracteres = 0;
    int  nombre_mots = 0;
    int  nombre_lignes = 0;
    Etat etat = Hors_d_un_mot;
    int c;

    for (c = getchar(); c != EOF; c = getchar())
    {
        switch (c)
        {
          case '\n':
            /* Separateur de lignes */
            nombre_lignes++;
            /* NOBREAK */

          case ' ':
          case '\t':
            /* Separateur de mots */
            if (etat == Dans_un_mot)
            {
                nombre_mots++;
                etat = Hors_d_un_mot;
            }
            break;
```

```
            default:
              /* Le caractere lu n'est pas un separateur */
              etat = Dans_un_mot;
              break;
          }
          nombre_caracteres++;
      }
      printf("%d ligne%s - %d mot%s - %d caractere%s\n",
             nombre_lignes, PLURIEL(nombre_lignes),
             nombre_mots, PLURIEL(nombre_mots),
             nombre_caracteres, PLURIEL(nombre_caracteres));
  }
```

compter.c

On peut comparer le résultat de l'exécution de la commande `compter` avec celui produit par la commande `wc` :

```
$ compter < compter.c
50 lignes - 112 mots - 1111 caracteres
$ wc compter.c
      50     112    1111 compter.c
$
```

4.2.3 Itérations

Il existe trois formes d'**itérations** : deux boucles *tant-que* et une boucle *pour*.

Boucles tant-que

Les deux boucles *tant-que* sont la boucle **while** et la boucle **do-while**. La syntaxe de ces instructions est :

> *instruction-tantque :* `while (`*expression*`)`
> *corps-de-boucle*
> *instruction-tantque :* `do`
> *corps-de-boucle*
> `while (`*expression*`);`

Le *corps de boucle* peut être indifféremment une instruction ou un bloc. La sémantique de ces instructions est la sémantique habituelle :

- **while** : tant que la valeur de l'expression de contrôle est non nulle, le corps de boucle est exécuté;
- **do** : le corps de boucle est exécuté tant que la valeur de l'expression de contrôle est différente de zéro.

Le programme suivant, `while.c`, illustre le comportement de la boucle `while` :

while.c

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

```
void usage(char *);

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

    if (argc != 2)
        usage(argv[0]);

    n = atoi(argv[1]);
    printf("La boucle while s'execute pour\n");
    while (n > 0)
    {
        printf("    n = %d\n", n);
        n --;
    }
}

void usage(char *s)
{
    fprintf(stderr, "Usage: %s nombre\n", s);
    exit(1);
}
```
while.c

Lorsque ce programme reçoit un nombre négatif ou nul en argument, la boucle n'est pas exécutée.

```
$ while 2
La boucle while s'execute pour
    n = 2
    n = 1
$ while 0
La boucle while s'execute pour
$
```

La boucle do est généralement moins utilisée. Elle est réservée aux situations où le corps de boucle doit être exécuté au moins une fois.

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

void usage(char *);

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

    if (argc != 2)
        usage(argv[0]);

    n = atoi(argv[1]);
    printf("La boucle do s'execute pour\n");
    do
    {
```

```
        printf("    n = %d\n", n);
        n--;
    }
    while (n > 0);
}

void usage(char *s)
{
    fprintf(stderr, "Usage: %s nombre\n", s);
    exit(1);
}
```
══════════════ do.c ══════

Contrairement à l'exemple précédent, la boucle est exécutée même lorsque l'argument de la commande est négatif ou nul.

```
$ do 2
La boucle do s'execute pour
    n = 2
    n = 1
$ do 0
La boucle do s'execute pour
    n = 0
$
```

Il est fréquent d'utiliser une expression à effet de bord comme expression de contrôle d'une boucle. Par exemple, itérer une instruction n fois s'écrit

```
while (n-- > 0)
    ⟨instruction⟩
```

Dans ce cas, l'utilisation de la variable `n` dans ⟨*instruction*⟩ n'est pas souhaitable. Pour illustrer cela, considérons le programme suivant :

══════ whileplusplus.c ══════════════
```
#include <stdio.h>
#include <stdlib.h>

static void usage(char *);

main(int argc, char *argv[])
{
    int i = 0;
    int n;

    if (argc != 2)
        usage(argv[0]);

    n = atoi(argv[1]);

    while(i++ < n)
        printf(" boucle pour (%d < %d)\n", i, n);
}
```

```
static void usage(char *s)
{
    fprintf(stderr, "Usage: %s nombre\n", s);
    exit(1);
}
```

whileplusplus.c

Comme on peut le voir sur l'exemple d'exécution suivant, la valeur de la variable ne vérifie pas systématiquement la condition de la boucle :

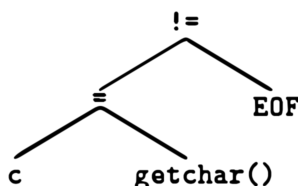
```
$ whileplusplus 3
 boucle pour (1 < 3)
 boucle pour (2 < 3)
 boucle pour (3 < 3)
$
```

Dans le test d'une boucle, il est préférable d'éviter d'effectuer une post-incrémentation ou une post-décrémentation sur une variable réutilisée dans le corps cette boucle.

L'utilisation d'une affectation dans l'expression de contrôle de la boucle est par contre relativement courante, comme dans cet exemple que nous avons déjà mentionné :

```
while ((c = getchar()) != EOF)
    ⟨instruction⟩
```

L'arbre de syntaxe de l'expression de contrôle est



Sa valeur est 1 si le caractère lu par `getchar` est différent de `EOF`, et 0 sinon. Par conséquent, la boucle est effectuée tant que le caractère lu n'est pas `EOF`. L'utilisation d'une expression à effet de bord est, dans ce cas, assez naturelle.

Boucle pour

La syntaxe de la boucle *pour* est :

boucle-pour : `for` ([⟨$expr_1$⟩]; [⟨$expr_2$⟩]; [⟨$expr_3$⟩])
⟨*corps-de-boucle*⟩

Le *corps de boucle* peut être indifféremment une instruction ou un bloc. L'expression ⟨$expr_1$⟩ est évaluée une seule fois, au début de l'exécution de la boucle. La boucle se déroule ensuite en répétant les actions suivantes :

1) si ⟨*expr₂*⟩ vaut zéro
   sortie de la boucle;
2) exécution de ⟨*corps-de-boucle*⟩;
3) évaluation de ⟨*expr₃*⟩.

Les trois expressions jouent donc le rôle respectivement d'expression d'initialisation, de test d'arrêt, et d'incrémentation. Par exemple

```
for (i=debut; i<=fin; i++)
```

doit se lire *"pour* `i` *variant de* `debut` *à* `fin` *par pas de 1, faire"*. Chacune des trois expressions est optionnelle. La construction

```
for (;;)
```

génère une boucle infinie[2].

Voici maintenant quelques exemples classiques d'utilisation de la structure de contrôle `for`. L'instruction

```
for (c = getchar(); c != '\n' || c != EOF; c = getchar())
    ;
```

défile les caractères lus sur l'entrée standard jusqu'au premier caractère `\n` rencontré. Si `p` est un pointeur de caractères, et `chaine` une chaîne de caractères, l'instruction

```
for (p = chaine; *p; p++)
    ;
```

place le pointeur `p` à la fin de la chaîne. Si `chaine` est une chaîne de caractères numériques (c'est-à-dire dont tous les caractères sont des chiffres), l'instruction

```
for (p = chaine, v = 0; isdigit(*p); v = 10*v + *p++ - '0')
    ;
```

calcule dans la variable entière `v` la valeur numérique de la chaîne. La fonction `isdigit`, qui teste si un caractère est un chiffre, est définie dans le fichier en-tête standard `ctype.h` (voir 7.2.1). Enfin, voici un exemple de programme utilisant des boucles imbriquées :

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

static void usage(char *);

main(int argc, char *argv[])
{
    int ligne, colonne;
    int valeur = 1;
    int dimension;

    if (argc != 2)
        usage(argv[0]);

    dimension = atoi(argv[1]);
    for (ligne = 0; ligne < dimension; ligne++)
    {
```

[2]Cette construction est très utilisée, la sortie de la boucle étant alors contrôlée par une instruction d'échappement.

```
        for (colonne = 0; colonne < ligne; colonne++)
            printf("   ");
        for (; colonne < dimension; colonne++, valeur++)
            printf("%2d ", valeur);
        printf("\n");
    }
}

static void usage(char *s)
{
    fprintf(stderr, "Usage: %s nombre\n", s);
    exit(1);
}
```
triangle.c

Par exemple, le triangle de dimension 5 est rempli avec les entiers de 1 à 15 :

```
$ triangle 5
 1  2  3  4  5
    6  7  8  9
      10 11 12
         13 14
            15
$
```

On notera qu'il suffit de récrire la boucle `for` la plus externe

```
for (ligne = dimension-1; ligne >= 0; ligne--)
```

pour obtenir à l'exécution le triangle suivant :

```
$ triangle 5
             1
          2  3
       4  5  6
    7  8  9 10
11 12 13 14 15
$
```

4.3 Instructions d'échappement

Les **échappements** sont des instructions permettant de rompre le déroulement séquentiel d'une suite d'instructions. La plus connue des instructions d'échappement, le branchement inconditionnel ou `goto`, est également celle qui possède la plus mauvaise réputation, son utilisation pouvant aller à l'encontre des règles de la programmation structurée. Il existe également des échappements compatibles avec ces règles, que nous appellerons les **échappements structurés**.

Alors que le branchement inconditionnel peut dégrader considérablement la lisibilité d'un programme, les échappements structurés, par contre, en facilitent l'écriture et la relecture de façon appréciable.

S'il est bien connu que ces instructions ne sont pas indispensables, et qu'on peut exprimer tout algorithme séquentiel au moyen de *tests* et de *boucles*, il est également bien connu des programmeurs que cela ne se fait pas sans mal; en particulier cela nécessite l'utilisation intensive de variables booléennes contrôlant le déroulement du programme (sorties de boucles, retours de fonctions...). De plus, il est souvent nécessaire d'imbriquer de nombreux tests les uns dans les autres, ce qui rend l'indentation des instructions assez pénible.

Les instructions d'échappement ne sont donc pas des gadgets. Elles allient efficacité et lisibilité, et leur utilisation facilite une programmation claire et structurée. Par conséquent, elles doivent être employées chaque fois que c'est possible.

4.3.1 Echappements structurés

La forme la plus générale d'échappement structuré est la sortie de n niveaux de blocs imbriqués (voir par exemple l'instruction `break` de l'interprète shell). Elle est partiellement implémentée en C sous la forme de deux instructions :

- **continue** : passage à l'itération suivante de la boucle englobante la plus proche;
- **break** : sortie de la boucle ou de l'aiguillage englobants le plus proche.

Une troisième façon d'effectuer un échappement est d'exécuter un `return` dans une fonction. Cette instruction, qui permet de sortir du bloc principal d'une fonction, est de nature légèrement différente des deux précédentes. En effet, le retour de fonction est une opération dynamique : l'adresse de l'instruction suivante à exécuter, c'est-à-dire l'adresse de retour, est contenue dans la pile d'exécution du processus. Par contre, les branchements provoqués par `break` et `continue` sont statiques : les adresses de branchements sont déterminées à la compilation.

La syntaxe de ces trois instructions est :

instruction-échappement : `continue;`
`break;`
`return` [*expression*] `;`

Instructions break et continue

Nous avons déjà présenté la sémantique de l'instruction `break` lorsqu'elle est utilisée dans un aiguillage. Dans le cas des boucles, les instructions `continue` et `break` se comportent comme des branchements à des étiquettes situées respectivement, juste avant et juste après la fin de la boucle. Si on considère la structure générale d'une boucle telle qu'elle est visualisée sur la figure 4.1, un `continue` dans le corps de la boucle provoque un branchement à l'étiquette et_1, et un `break` à l'étiquette et_2.

Un des intérêts de l'instruction `continue`, à savoir d'éviter l'imbrication multiple de tests, est illustré par la commande `maxmot` de l'exemple suivant. Celle-ci imprime le mot le plus long lu sur son entrée standard. Dans le cas où plusieurs mots sont de même longueur, elle imprime le premier rencontré. Le test sur la nature de chaque caractère utilise la bibliothèque `ctype` (voir 7.2.1).

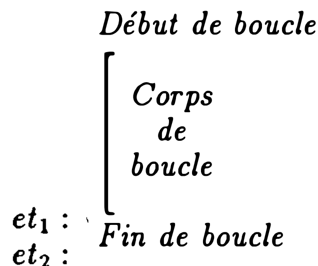


Figure 4.1: Structure générale d'une boucle

On remarquera que l'utilisation de l'instruction `continue`, ainsi que celle de l'instruction `break`, incite à construire le programme de façon à traiter d'abord les cas triviaux afin de les éliminer. Pour se persuader de l'intérêt de l'instruction `continue` on pourra s'exercer à récrire cette fonction en se restreignant à des tests et des boucles.

```
maxmot.c
#include <stdio.h>
#include <ctype.h>

#define TXMOT   256

static int dans_un_mot(char);

main()
{
    char le_mot[TXMOT+1];
    char le_candidat[TXMOT+1];
    char *fin_mot = le_candidat;
    int longueur = 0;
    int longueur_max = 0;
    char c;

    for (c = getchar(); c != EOF; c = getchar())
    {
        if (dans_un_mot(c))
        {
           /*--- Lecture du mot courant */
            if (longueur == TXMOT)
                continue;
            longueur++;
            *fin_mot++ = c;
            continue;
        }
        if (longueur == 0)
            /* Defilement d'une suite de separateurs */
            continue;
        if (longueur > longueur_max)
        {
            /* Mot courant plus long que les precedents */
            *fin_mot = '\0';
            strcpy(le_mot, le_candidat);
```

```
            longueur_max = longueur;
        }
        fin_mot = le_candidat;
        longueur = 0;
    }
    printf("%s", le_mot);
}

/*
  Retourne vrai si le caractere recu en parametre
  est un caractere d'un mot.
  */
static int dans_un_mot(char c)
{
    return isalnum(c) || c == '_';
}
```
maxmot.c

L'exemple suivant utilise des mécanismes classiques des *shells* UNIX : la commande `maxmot` reçoit sur son entrée standard son propre fichier source, et le résultat qu'elle écrit sur sa sortie standard est réutilisé pour construire le message à afficher.

```
$ echo 'maxmot < maxmot.c' est le mot le plus long de maxmot.c
longueur_max est le mot le plus long de maxmot.c
$
```

L'instruction `break` est très utile pour programmer une itération comportant plusieurs conditions d'arrêt, comme la recherche du premier élément d'une liste vérifiant une propriété donnée. Nous allons en donner deux versions.

La première version est un parcours de chaîne de caractères avec arrêt sur le caractère `\n` ou en fin de chaîne; dans le premier cas, le caractère `\n` est remplacé par le caractère `\0`.

```
for (p = chaine; *p; p++)
    if (*p == '\n')
    {
        *p = '\0';
        break;
    }
```

Sans l'instruction `break`, il aurait été nécessaire d'utiliser un booléen pour gérer la sortie de boucle :

```
for (p = chaine, continuer = 1; *p && continuer; p++)
    if (*p == '\n')
    {
        *p = '\0';
        continuer = 0;
    }
```

La seconde version est la recherche dans un vecteur du premier élément vérifiant une condition donnée, par exemple dont la valeur est supérieure à celle de son successeur :

```
for (baisse=INDEFINI, mois=JANVIER; mois<DECEMBRE; mois++)
    if (resultat[mois] > resultat[mois+1])
    {
        baisse = mois + 1;
        break;
    }
```

L'instruction return

L'instruction `return` provoque la sortie de la fonction qui la contient. Par exemple, lors de l'exécution de la fonction

```
void initialiser_champs(struct champs *ch)
{
    if (!permission(ch, ECRITURE))
        return;
    ...
}
```

si l'expression `permission(ch, ECRITURE)` vaut *faux*, le retour de fonction est effectué immédiatement, sinon l'instruction suivante est exécutée. Si le mot-clé `return` est suivi d'une expression, cette expression est évaluée et son résultat est retourné par la fonction. Par exemple, la fonction `permission` de l'exemple précédent peut s'écrire :

```
int permission(struct champs *ch, short mode)
{
    return ch->permissions & mode;
}
```

Le type de l'expression retournée et le type de la fonction doivent être compatibles. Soit, par exemple, le type `liste` défini par :

```
struct liste
{
    struct liste *suivant;
    int numero;
};

typedef struct liste liste;
```

La compilation de la fonction :

```
element_suivant(liste *element)
{
    if (element == NIL)
        return NIL;
    return element->suivant;
}
```

provoque l'émission d'un message d'erreur de la forme :

```
liste.c: In function element_suivant:
liste.c:14: warning: return of integer from pointer
```

```
                    lacks a cast
```

Avec le compilateur C traditionnel, on obtient le message :

```
"liste.c",line 14: warning: illegal combination of pointer and
                             integer, op RETURN
```

En effet, comme nous l'avons vu page 19, une fonction implicitement est de type `int`. L'instruction

```
return element->suivant
```

correspond à l'affectation d'une valeur de type `liste *` dans un objet de type entier. La définition correcte de la fonction `element_suivant` est :

```
liste *element_suivant(liste *element)
{
    ...
```

Voici, pour terminer, l'exemple d'une recherche par dichotomie dans une liste d'entiers triée. Cette liste est implémentée sous la forme d'un vecteur. La recherche est effectuée entre les positions `debut` et `fin`. On suppose que `debut` est positif. Si la fonction retourne un nombre positif, il s'agit de la position dans la liste de la valeur cherchée. Si elle retourne un nombre négatif n, la valeur n'est pas dans la liste, et $-n$ est la position qu'elle devra occuper si on l'y insère.

dichotomie.c

```
#include "dichotomie.h"

int recherche(int debut, int fin,
              int valeurs[], int valeur_cherchee)
{
    int milieu;

    for (;;)
    {
        if (fin < debut)
        {
            /* La valeur cherchee n'est pas dans la liste */
            return -debut;
        }
        milieu = (debut + fin) / 2;
        if (valeurs[milieu] == valeur_cherchee)
            /* La valeur cherchee a ete trouvee */
            return milieu;

        if (valeurs[milieu] < valeur_cherchee)
            /* Recherche dans la partie haute de la liste */
            debut = milieu + 1;
        else
            /* Recherche dans la partie basse de la liste */
            fin = milieu - 1;
    }
}
```

dichotomie.c

Encore une fois, on remarquera le style consistant à traiter en premier les cas particuliers. Cela permet de minimiser les imbrications de blocs, et par conséquent, d'accroître la lisibilité du programme.

4.3.2 Branchement inconditionnel

Le **branchement inconditionnel** est à utiliser avec la plus grande prudence. Cependant, il est parfois utile pour simuler des échappements structurés non implémentés en C (essentiellement la sortie de plusieurs boucles imbriquées).

Un branchement s'effectue sur une **instruction étiquetée**; la syntaxe d'une instruction étiquetée est

instruction-étiquetée : identificateur: *instruction*

L'instruction pouvant être une instruction vide. La syntaxe du branchement est

instruction-branchement : `goto` *étiquette* ;

L'étiquette doit être située dans la même fonction que le branchement.

Le branchement inconditionnel ne devrait être utilisé que lorsque les instructions `break` et `continue` sont insuffisantes, par exemple dans le cas de `break`, pour sortir d'un aiguillage imbriqué dans une boucle. On peut comparer les versions avec et sans `goto` :

```
    for (;;)
    {
        char c;

        switch(c = getchar())
        {
          case QUIT :
              goto end_for;

          case ...

        }
    }
  end_for:
    ;
```

```
for (;;)
{
    char c;

    if ((c = getchar()) == QUIT)
        break;
    switch (c)
    {
      case ...

    }
}
```

Une autre situation concerne la sortie de plusieurs boucles imbriquées; c'est le cas de l'exemple suivant, présentant le parcours d'une liste de listes avec arrêt sur le premier élément vérifiant une condition donnée :

```
    for (jour = 0; jour < 7; jour++)
          for (heure = 1; heure < 24; heure++)
              if (semaine[jour][heure]<semaine[jour][heure-1])
                  goto baisse;
  baisse:
    ;
```

On peut éviter ici l'utilisation du branchement en plaçant l'instruction dans une fonction, dont on sort au moyen d'un `return` :

```
void recherche(int *j, int *h)
{
    int jour, heure;

    for (jour = 0; jour < 7; jour++)
        for (heure = 1; heure < 24; heure++)
            if (semaine[jour][heure]<semaine[jour][heure-1])
            {
                *j = jour;
                *h = heure;
                return;
            }
    *j = jour;
    *h = heure;
}
```

mais la gestion des paramètres par référence alourdit l'écriture. Il est plus simple d'écrire :

```
void recherche(int *j, int *h)
{
  int jour, heure;

  for (jour = 0; jour < 7; jour++)
        for (heure = 1; heure < 24; heure++)
            if (semaine[jour][heure]<semaine[jour][heure-1])
                goto fin_de_boucle;

fin_de_boucle:
  *j = jour;
  *h = heure;
}
```

Une autre situation classique est la gestion d'exceptions, avec interruption du traitement (`goto erreur`) ou reprise (`goto debut`). Tous ces exemples d'utilisation du branchement inconditionnel sont construits sur le même modèle : la sortie de plusieurs blocs imbriqués. Par contre, on respectera la règle suivante :

Il ne faut jamais utiliser un goto pour se déplacer dans un bloc ou pour rentrer dans un bloc de niveau inférieur.

4.3.3 Fonctions provoquant une rupture de séquence

La terminaison d'un programme peut être forcée au moyen de la fonction de bibliothèque `exit` dont le prototype :

```
void exit(int)
```

est défini dans le fichier en-tête `<stdlib.h>`. Sous UNIX, cette fonction met en œuvre l'appel système **_EXIT(2)** provoquant la terminaison du processus.

Le paramètre de l'appel `exit` est le code de terminaison du programme; la valeur 0 signifie *terminaison sans erreur*, et une valeur non nulle e (de préférence 1) signifie *terminaison sur erreur.*

Un exemple très classique d'utilisation de la fonction `exit` est celui de la détection d'une erreur dans la liste des arguments d'une commande. On pourra prendre l'habitude de définir une fonction imprimant la description de la commande et provoquant une terminaison sur erreur. Traditionnellement, cette fonction est appelée `usage`, le message d'erreur qu'elle émet étant de la forme

`Usage:` *nom-de-la-commande* ...

Il y a de nombreux exemples de définition de la fonction `usage` dans cet ouvrage; la forme générale de cette fonction est :

```
void usage(char *cmd)
{
    fprintf(stderr, "Usage: %s arguments", cmd);
    exit(1);
}
```

L'appel est effectué avec l'expression `argv[0]` comme paramètre effectif, par exemple :

```
if (nombre d'argument incorrect)
    usage(argv[0]);
```

Dans tout programme, on prendra l'habitude de définir, dans le fichier où est traitée la récupération des arguments, la fonction d'erreur `usage` *qui imprime le guide d'utilisation de la commande et provoque la terminaison avec un code d'erreur.*

Un autre exemple classique d'utilisation de l'appel `exit` est la sortie d'une application interactive. Voici le schéma de la fonction `fin` effectuant cette opération :

```
void fin()
{
    Gestion de la terminaison:
        - fermeture des fichiers ouverts,
        - restauration du tty,
        - ...
    exit(0);
}
```

On notera qu'`exit` est une fonction et non une instruction. Pour le compilateur, rien ne la différencie d'une autre fonction. Pour mieux comprendre la différence entre la fonction `exit` et l'instruction `return`, considérons le programme suivant :

```
====== return_exit.c ======
void f_return()
{
    return;
    printf("Jamais\n");
}

void f_exit()
```

```
{
    exit(0);
    printf("Jamais\n");
}
```

return_exit.c

Sa compilation par un compilateur C traditionnel produit le message d'avertissement :

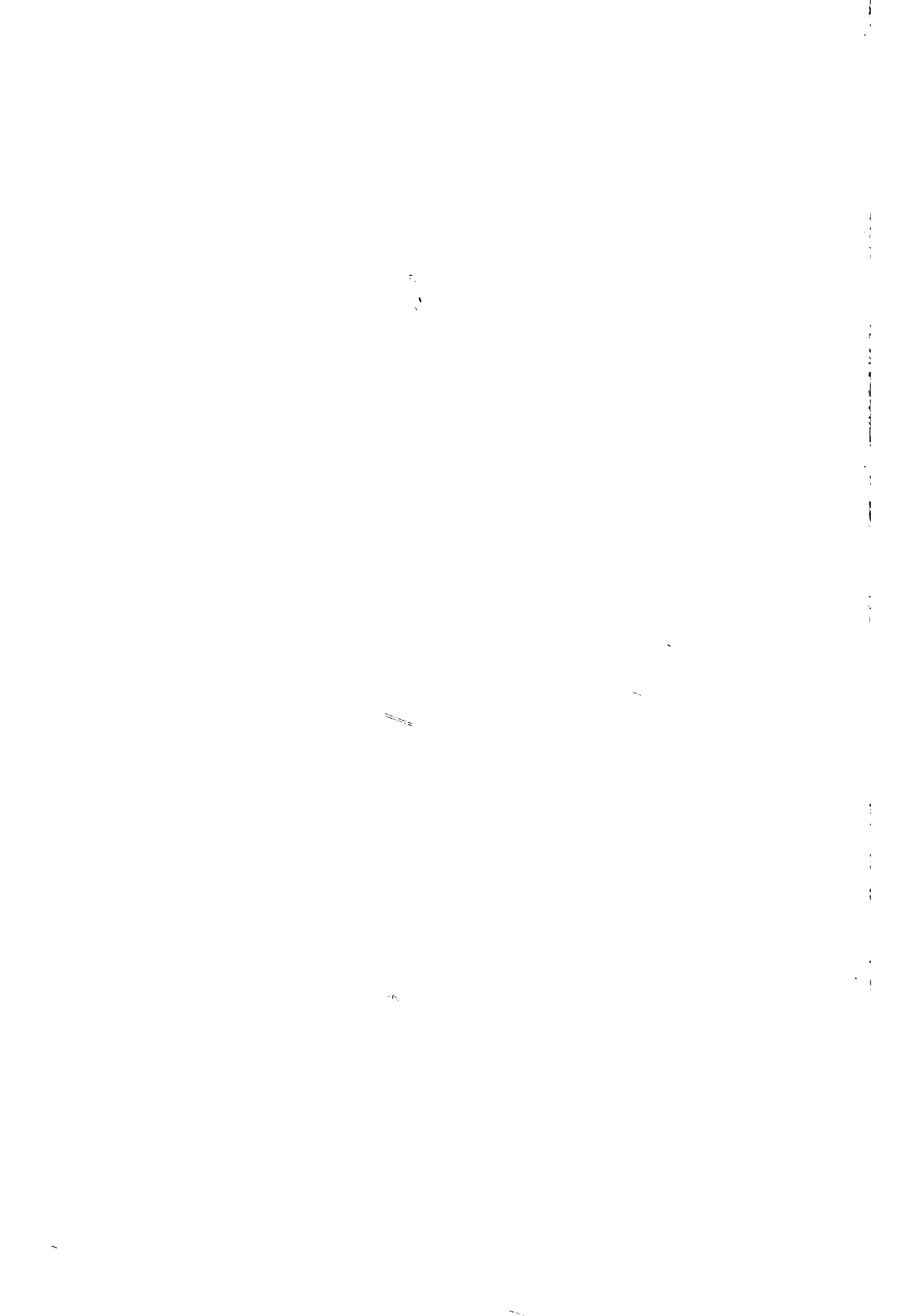
```
$ cc -c return_exit.c
"return_exit.c", line 4: warning: statement not reached
```

Aucune des deux instructions `printf` ne sera jamais atteinte, mais le compilateur ne peut le détecter que dans le premier cas.

Il existe une autre façon de contrôler l'exécution d'un programme, consistant à sauvegarder la pile d'exécution du processus en train de s'exécuter, puis à restaurer ce contexte. Ce mécanisme est géré par les fonctions `setjmp` et `longjmp` présentées en 7.9.

Deuxième partie

Environnement et méthodes de programmation



Chapitre 5

L'utilisation du préprocesseur

Lorsqu'on parle de la compilation d'un programme C, on sous-entend en général deux traitements distincts :

1) le **prétraitement** du programme par le **préprocesseur**;
2) la compilation proprement dite du fichier résultant de ce prétraitement par le compilateur.

Le préprocesseur effectue le prétraitement du programme source en exécutant des instructions spéciales appelées **directives**. Ces directives sont identifiées par un dièse (caractère #) en première colonne; elles peuvent se continuer sur plusieurs lignes, chaque ligne à continuer étant terminée par un anti-slash (caractère \). Les lignes qui ne sont pas des directives sont recopiées telles quelles par le préprocesseur. Le préprocesseur offre quatre fonctionnalités différentes :

- la définition et l'expansion de **pseudo-constantes** et **pseudo-fonctions**,
- l'inclusion de fichiers,
- la compilation conditionnelle,
- le pilotage des messages d'erreur.

Le préprocesseur du langage C est l'analogue d'un macro-assembleur pour un langage machine.

5.1 Définition de pseudo-fonctions

5.1.1 Pseudo-constantes

Directives define et undef

Une pseudo-constante est un symbole, défini au moyen d'une directive, et dont la valeur est une chaîne de caractères. Cette définition s'effectue au moyen de la directive `define` dont la syntaxe est

directive-définition : `#define` *identificateur* *chaîne-de-substitution*

Un symbole de pseudo-constante est composé de lettres et de chiffres et commence par une lettre. Le caractère _ est une lettre. L'identificateur de la

pseudo-constante et la chaîne de substitution doivent être séparés par au moins un séparateur, c'est-à-dire une espace ou une tabulation.

La chaîne peut contenir n'importe quel caractère et se termine sur le caractère *newline* de fin de ligne, sauf si le caractère précédant *newline* est le caractère \; dans ce cas, la chaîne se poursuit sur la ligne suivante.

Durant le prétraitement d'un fichier, le préprocesseur recherche les occurrences des pseudo-constantes définies, c'est-à-dire les mots correspondant à un symbole de pseudo-constante qui ne sont pas sous-mots d'un mot plus long. Chaque occurrence d'une pseudo-constante trouvée est remplacée par sa valeur.

Par exemple, la directive

```
#define TAILLE_VECTEUR   255
```

définit la pseudo-constante TAILLE_VECTEUR et lui attribue pour valeur le mot 255. Lorsqu'il rencontre cette directive, le préprocesseur remplace dans la suite du fichier source toutes les occurrences du mot TAILLE_VECTEUR par le mot 255. Ainsi, l'instruction

```
double vecteur[TAILLE_VECTEUR]
```

est récrite en

```
double vecteur[255]
```

Par contre, le mot TAILLE_VECTEUR_PROPRE n'est pas récrit en 255_PROPRE car le mot TAILLE_VECTEUR est, dans ce cas, un sous-mot de l'identificateur TAILLE_VECTEUR_PROPRE.

Par défaut, une définition demeure valide jusqu'à la fin du fichier où elle apparaît. Il est possible d'annuler une définition au moyen de la directive **undef**. Par exemple la directive

```
#undef TAILLE_VECTEUR
```

annule la définition de la pseudo-constante TAILLE_VECTEUR pour toutes les lignes qui suivent.

Mise en garde

Il faut toujours utiliser ces définitions dans le seul but d'accroître la modularité et la lisibilité d'un programme. On rencontre cependant de nombreux exemples d'utilisation allant à l'encontre de ces principes. Considérons le fichier "defs.h" suivant :

```
defs.h
#define TRUE     1
#define FALSE    0
#define bool     int

#define begin    {
#define end      }
#define then
defs.h
```

Les deux premières directives définissent les pseudo-constantes TRUE et

FALSE respectivement comme les chaînes de caractères 1 et 0. La troisième indique que toute occurrence de la pseudo-constante bool doit être récrite en int. Ces trois directives correspondent à des définitions assez classiques de pseudo-constantes. On préfèrera cependant en C standard la solution suivante : le fichier booleen.h contient la définition de deux constantes entières vrai et faux et du type bool :

```
booleen.h
#ifndef BOOLEEN_H
#define BOOLEEN_H

typedef int bool;

extern const bool vrai;
extern const bool faux;

#endif  /* BOOLEEN_H */
booleen.h
```

et le fichier booleen.c l'initialisation de ces constantes :

```
booleen.c
#include "booleen.h"

const bool vrai = 1;
const bool faux = 0;
booleen.c
```

On pourra encore utiliser cette autre façon de faire:

```
autre_booleen.h
#ifndef AUTRE_BOOLEEN_H
#define AUTRE_BOOLEEN_H

typedef enum { faux = 0, vrai = 1 } booleen;

#endif  /* AUTRE_BOOLEEN_H */
autre_booleen.h
```

Les trois dernières définitions du fichier defs.h ont pour seul but de transformer la syntaxe du langage. Elles indiquent que les chaînes begin et end doivent être récrites respectivement en { et }, et que la chaîne then doit être récrite en un mot vide, c'est-à-dire purement et simplement effacée. Voyons maintenant comment un programme utilisant ces définitions est récrit par le préprocesseur. Il s'agit du programme redefinitions.c suivant :

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

#include "defs.h"

main()
begin
```

```
    bool test;

    test = 1 < 2;
    if (test == FALSE)
    then
        printf("Le compilateur C genere des erreurs\n");
end
```

redefinitions.c

L'option de compilation -E permet de visualiser le résultat du prétraitement d'un fichier source. Seul le préprocesseur est invoqué, le résultat étant envoyé vers la sortie standard :

```
$  gcc -E redefinitions.c
# 1 "redefinitions.c"
main()
   {
        int test;

    test = 1 < 2;
    if (test ==    0)
         printf("Le compilateur C genere des erreurs\n");
     }
$
```

Excepté la première ligne qui est utilisée par le compilateur pour la génération des messages d'erreur (nom du fichier et numéro de ligne), le reste du programme est du langage C pur.

Le programme ainsi généré est certes tout à fait correct, mais le source n'est plus vraiment un programme en langage C. Parmi les *classiques* de ce genre de pratique, on trouve également

```
#define private    static
#define public     extern
#define ever       ;;
#define repeat     do
#define until(e)   while (!e)
```

Ce genre de définition nuit gravement à la lisibilité d'un programme. De plus, l'auteur de telles redéfinitions s'expose à des erreurs difficiles à déceler lorsque les définitions ne sont pas consistantes (ce qui est souvent le cas...). En voici un exemple. On définit au moyen de pseudo-constantes les unités K et MEGA, afin de pouvoir former des expressions comme 10 K, ou 1 MEGA. Cette définition consiste à produire respectivement une multiplication par 2^{10} et 2^{20}. Cela s'écrit:

```
#define K       *(1UL << 10)
#define MEGA    *(1UL << 20)
```

Malheureusement, on sera vite tenté d'écrire 1K ou 1MEGA, ce qui produira une erreur (pourquoi ?). La bonne façon d'écrire ces définitions est de les définir comme des constantes

```
═══════ unites.h ═══════════════════════════
#ifndef UNITES_H
#define UNITES_H

#define K    (1UL << 10)
#define MEGA (1UL << 20)

#endif  /* UNITES_H */
═══════════════════════════════ unites.h ═══════
```

et de les utiliser dans des expressions

```
═══════ test_unites.c ═══════════════════════
#include <stdio.h>
#include "unites.h"

main()
{
    printf("%d, %d, %d, ... %d\n", 1*K, 2*K, 3*K, 1*MEGA);
}
═══════════════════════════ test_unites.c ═══════
```

L'exécution de ce programme produit le résultat :

```
$ test_unites
1024, 2048, 3072, ... 1048576
$
```

Fort heureusement cette pratique, qui a eu son heure de gloire[1], tend à disparaître. Cela ne veut pas dire pour autant qu'il faut s'interdire toute définition d'ordre syntaxique. Mais il faut toujours les faire pour introduire une nouvelle construction, et non pour en renommer une déjà présente.

Quelques erreurs classiques

Un autre danger des macro-définitions est lié au fait que le préprocesseur ignore la syntaxe du langage C lorsqu'il réalise les substitutions. Par exemple, la directive

```
#define max   100
```

a pour effet de remplacer toutes les occurrences de `max` par `100`. Si on définit par mégarde, dans la suite du fichier, une variable `max`, par exemple

```
int max;
```

le préprocesseur remplacera cette ligne de déclaration par la ligne :

```
int 100;
```

Ce cas est illustré par le fichier `errdef.c` suivant :

---

[1] Un des exemples parmi les plus illustres est le source original de l'interprète de commandes shell écrit par Steve Bourne, rendu totalement illisible par une quarantaine de directives redéfinissant intégralement la syntaxe du langage.

errdef.c
```
#define max 100

int maximum(int v[])
{
    int i;
    int max;

    for (i=0; v[i] > 0; i++)
    {
        if (v[i] > max)
            max = v[i];
    }
    return max;
}
```
errdef.c

Pendant la compilation de ce fichier, le compilateur signale une erreur de syntaxe sur une ligne à priori correcte.

```
$ gcc -g -c errdef.c
errdef.c: In function 'maximum':
errdef.c:6: parse error before '100'
errdef.c:11: invalid lvalue in assignment
$
```

Bien entendu, sur un exemple de petite taille comme celui-ci, l'erreur est assez facile à identifier. Cependant, la directive de définition peut être située dans un autre fichier, et introduite dans celui-ci par une directive d'inclusion (voir 5.2). Nous avons là un exemple d'erreur qui peut s'avérer difficile à diagnostiquer. On peut facilement éviter ce type d'erreurs en différenciant lexicalement les pseudo-constantes des identificateurs de variables et de constantes. On appliquera rigoureusement le principe suivant :

Les identificateurs de type, de constante, de variable, et de fonction doivent contenir au moins une lettre minuscule; les symboles de pseudo-constantes et de pseudo-fonctions doivent être formés exclusivement au moyen de lettres majuscules, de chiffres, et du caractère souligné.

Une autre erreur assez classique consiste à terminer une directive de définition par un point-virgule, par exemple :

```
# define TMAX     100;
```

définition qui provoquera la réécriture de la ligne

```
int tableau[TMAX];
```

en

```
int tableau[100;];
```

Cette faute d'inattention relativement courante peut parfois être longue à détecter. Aussi peut-il être fructueux, lorsqu'on se trouve confronté à des erreurs de syntaxe *"incompréhensibles"*, de passer soigneusement en revue la liste

des `define` du fichier incriminé, en y recherchant d'éventuels points-virgules illégaux.

5.1.2 Définition à la compilation

Il est possible de définir des pseudo-constantes lors de la phase de compilation, au moyen de l'option `-D`. Cette option s'utilise en concaténant à sa suite le nom de la pseudo-constante à définir. Par exemple, la définition de la variable `NDEBUG` s'écrit `-DNDEBUG`. L'exécution de

```
gcc -g -DNDEBUG main.c l.c
```

est équivalente à celle de `gcc -g main.c l.c` avec, en tête des fichiers `main.c` et `l.c`, la ligne `#define NDEBUG` Il est également possible de préciser la valeur de la constante en rajoutant le symbole `=` suivi de cette valeur. Par exemple, l'option

```
-DTAILLE_BLOC=1024
```

est équivalente à la définition `#define TAILLE_BLOC 1024`

5.1.3 Pseudo-constantes prédéfinies

Cinq pseudo-constantes sont prédéfinies dans la norme ANSI :

- `__FILE__` et `__LINE__` : nom du fichier et numéro de la ligne courants;
- `__DATE__` et `__TIME__` : date et heure de la compilation;
- `__STDC__` : définie si le compilateur se conforme à la norme ANSI.

Les deux premières existent dans la plupart des implémentations du langage C. Dans l'exemple suivant, la ligne 3 fait afficher la valeur de ces pseudo-constantes :

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

main()
{
    printf("Ligne %d du fichier %s\n", __LINE__, __FILE__);
}
LINEFILE.c
```

L'exécution de ce programme produit le résultat suivant :

```
$ gcc -g LINEFILE.c -o LINEFILE
$ LINEFILE
Ligne 3 du fichier LINEFILE.c
$
```

On trouvera un exemple d'utilisation de la pseudo-constante `__STDC__` à la page 165. Les pseudo-constantes `__DATE__` et `__TIME__` peuvent être utilisées pour étiqueter une version d'un logiciel. L'exemple suivant illustre la construction d'une chaîne de caractères contenant la date et l'heure de la compilation.

```
======== version.c ========================================
#include <stdio.h>

#define FORMAT   "** Version 1.37 (%s, %s) **"
#define TVERS    (sizeof FORMAT - 4 + sizeof __DATE__ \
                  + sizeof __TIME__ + 1)

static char version[TVERS];

main()
{
    sprintf(version, FORMAT, __DATE__, __TIME__);

    puts(version);
}
======================================== version.c ========
```

Voici une compilation et une exécution de ce programme précédées par l'affichage de la date courante.

```
$ date
Fri Jul 14 18:17:07 MET DST 1989
$ gcc -g version.c -o version
$ version
** Version 1.37 (Jul 14 1989, 18:17:11) **
$
```

En voici maintenant une seconde, quelques années plus tard...

```
$ date
Wed Jul 14 15:36:50 MET DST 1993
$ make version
gcc -g  -o version version.c
$ version
** Version 1.37 (Jul 14 1993, 15:36:54) **
$
```

5.1.4 Pseudo-fonctions

Paramètres de pseudo-fonctions

Le mécanisme de substitution de la directive `define` est paramétrable. On appellera une définition paramétrée une **pseudo-fonction**. La syntaxe générale d'une définition de pseudo-fonction est

directive-définition: `#define` *identificateur*(*paramètre,* `...`) *corps*

Le corps de la pseudo-fonction est une suite quelconque de caractères. Lors de la substitution, toute occurrence d'un paramètre dans le corps de la pseudo-fonction est remplacée par l'argument correspondant. Par exemple, la pseudo-

fonction *valeur absolue* peut être définie de la façon suivante :

```
#define ABS(x) x>0 ? x : - x
```

Toute occurrence d'une expression de la forme ABS(⟨*mot*⟩) sera récrite en

⟨*mot*⟩>0 ? ⟨*mot*⟩ : - ⟨*mot*⟩

Lorsqu'une pseudo-fonction est utilisée avec un nombre d'arguments incorrect, il y a émission d'un message d'erreur du type :

```
errabs.c:9: too many (2) args to macro 'ABS'
```

Avec le préprocesseur *GNU*, l'utilisation d'une pseudo-fonction sans argument ne génère pas de réécriture, et ne déclenche pas de message d'erreur. D'autres préprocesseurs peuvent, soit générer un message d'erreur:

```
errabs.c: 10: ABS: argument mismatch
```

soit effectuer la réécriture en considérant que le paramètre est le mot vide.

Mécanisme de substitution

Le mécanisme mis en œuvre lors du traitement d'une pseudo-fonction opère sur des expressions parenthésées. Cependant, une fois la substitution effectuée, la structure syntaxique de l'expression récrite est perdue. Par exemple, l'expression ABS(x+y) est remplacée par x+y>0 ? x+y : - x+y. On peut voir sur cet exemple deux dangers de ce mécanisme :

- les substitutions peuvent provoquer une multiplication du code et des calculs effectués; en effet, l'expression x+y est ici évaluée deux fois;
- la structure syntaxique de la définition peut être perdue; dans le cas de ABS, l'expression conditionnelle générée par le préprocesseur calcule, dans le cas où x+y est négatif, la valeur de -x+y et non celle de -(x+y) comme on aurait pu s'y attendre.

Un autre exemple classique, illustrant le défaut de structuration de la directive de définition, est la pseudo-fonction CARRE définie par

```
#define CARRE(x)   x*x
```

En effet, l'expression CARRE(x+1) est récrite en x+1*x+1 de valeur *2x+1*. La bonne façon de définir les pseudo-fonctions ABS et CARRE est :

```
#define CARRE(x)   ((x)*(x))
#define ABS(x)     ((x) > 0 ? (x) : -(x)
```

De cette façon, le motif CARRE(x+1) est expansé en

```
((x+1)*(x+1))
```

et ABS(x+y) en

```
((x+y) > 0 ? (x+y) : -(x+y))
```

D'une manière générale, on respectera la règle suivante :

Dans une définition de pseudo-fonction, toute occurrence d'un paramètre dans le corps de la définition doit être placée entre parenthèses.

Une autre erreur classique consiste à laisser un séparateur entre le nom d'une pseudo-fonction et la parenthèse ouvrante. Par exemple, la définition

```
#define CARRE (x)   ((x)*(x))
```

associe le mot CARRE à la chaîne

```
(x)   ((x)*(x))
```

Par conséquent, la suite de caractères

```
CARRE (p1)
```

est récrite en

```
(x)   ((x)*(x)) (p1)
```

L'erreur suivante est provoquée par l'utilisation d'une expression à effet de bord en paramètre d'une pseudo-fonction. L'expression

```
CARRE(x++)
```

est remplacée par l'expression

```
((x++)*(x++))
```

dont l'évaluation génère deux incrémentations de la variable x.

De façon plus générale, ce type d'erreur apparaît lorsque la pseudo-fonction produit une duplication de code. L'un des intérêts des pseudo-fonctions étant de permettre de remplacer un morceau de code complexe par une expression plus simple, on doit pouvoir ignorer la façon dont est écrite une pseudo-fonction. On adoptera donc la règle suivante :

⊘ *Il ne faut jamais utiliser une expression à effet de bord en paramètre d'une pseudo-fonction. Comme il n'est pas toujours possible de distinguer une pseudo-fonction d'une fonction, on pourra étendre ce principe à l'utilisation des fonctions véritables.*

On a d'ailleurs vu en 3.4 que les expressions à effet de bord peuvent également poser des problèmes lorsqu'elles sont utilisées en paramètre de fonction, ce qui constitue une raison supplémentaire d'étendre cette restriction aux paramètres des fonctions véritables.

Expressions ou instructions

Il est préférable, dans la mesure du possible, de définir une pseudo-fonction de telle façon que le résultat de son expansion soit une expression; elle a ainsi le même statut syntaxique qu'une fonction véritable. Par exemple, des deux définitions :

```
#define ECRIRE(fd,i)     (sprintf(buf,"%05d",(i)),\
                          write(fd,buf,strlen(buf)))
```

et

```
#define ECRIRE(fd,i)     sprintf(buf,"%05d",(i));\
                         write(fd,buf,strlen(buf))
```

on préfèrera la première. En effet, le motif ecrire(fd,n) est récrit en l'expression

```
(sprintf(buf,"%05d",(i)), write(fd,buf,strlen(buf)))
```

dont la valeur est le nombre de caractères écrits par la fonction `write` (voir 10.2.2). On peut, par conséquent, former indifféremment les instructions suivantes :

```
ECRIRE(fd, n);
nbcar = ECRIRE(fd,n);
for (i=0; i<9; ECRIRE(fd, i), i++)
    ;
```

Avec la seconde définition, seule la première instruction est correcte.

Fonctions ou pseudo-fonctions

On peut parfois hésiter dans une implémentation entre l'utilisation d'une pseudo-fonction et celle d'une fonction véritable. Le choix d'utiliser une pseudo-fonction peut être justifié par plusieurs raisons, dont la plus immédiate est liée l'efficacité du programme, le code expansé étant généralement plus rapide que celui de l'appel de fonction. Cependant, cet argument tend à s'effacer face à la puissance sans cesse croissante des machines et à l'efficacité du traitement des appels de fonction et des empilements/dépilements de contexte. Lorsque c'est possible, il est toujours préférable de choisir la fonction, qui offre un meilleur contrôle (typage des paramètres, non duplication du code...).

Il existe plusieurs situations où les pseudo-fonctions sont inévitables, parmi lesquelles :

- la manipulation de constructions qui ne sont pas de première catégorie[2] comme les types (on en trouvera un exemple page 171, avec un module générique de pile);
- le passage de paramètres par nom (c'est le cas de la construction `va_start` présentée en 7.7);
- l'utilisation de fonctionnalités spécifiques au préprocesseur, comme la génération ou la concaténation de chaînes (voir page 158);
- l'utilisation de constructions du préprocesseur, comme les pseudo-constantes prédéfinies;

Voici un exemple illustrant le dernier cas. Il s'agit d'une fonction de contrôle provoquant un arrêt du programme avec émission d'un message d'erreur lorsqu'une condition est vérifiée. La condition et le message sont fournis en paramètres, et la fonction imprime le nom du fichier et le numéro de ligne où s'est produit l'événement provoquant la terminaison.

exit_if.h

```
#ifndef EXIT_IF_H
#define EXIT_IF_H

#include <stdio.h>

#define EXIT_IF(expr,mess) \
    ((expr) \
```

[2]On appelle généralement *construction de première catégorie* (ou encore de *première classe*) d'un langage de programmation, une construction manipulable explicitement dans ce langage, c'est-à-dire comme argument d'une affectation, et comme paramètre ou valeur de retour d'une fonction.

```
      ? (fprintf(stderr, ">EXIT:%s:%d: %s\n",\
                 __FILE__, __LINE__ , mess), exit(1))\
      : 0)

#endif  /* EXIT_IF_H */
```
exit_if.h

Cette fonction peut être très utile pour traiter les erreurs irrécupérables, comme par exemple un défaut de mémoire lors d'une allocation dynamique. Le programme suivant permet de tester la pseudo-fonction EXIT_IF.

t_exit.c

```
#include "exit_if.h"

main()
{
    EXIT_IF(1, "test de sortie par EXIT_IF");
}
```

t_exit.c

Lors de son exécution, il produit le résultat suivant :

```
$ t_exit
>EXIT:t_exit.c:5: test de sortie par EXIT_IF
$
```

Génération de chaînes

Le préprocesseur du langage C ANSI permet de transformer un paramètre de pseudo-fonction en chaîne de caractères. Pour cela, l'identificateur du paramètre est précédé, dans le corps de la définition, du symbole #. Considérons par exemple la définition suivante

```
#define IPRINT(x) printf("%s = %d\n", #x, x)
```

Le motif IPRINT(2*3) est récrit par le préprocesseur ANSI en

```
printf("%s = %d\n", "2*3", 2*3)
```

Par conséquent, l'exécution de l'instruction IPRINT(2*3); produit l'affichage

```
2*3 = 6
```

Concaténation de paramètres

Il est possible en C ANSI de former un mot à partir des paramètres d'une pseudo-fonction. On utilise pour cela l'opérateur de concaténation ##. Par exemple, la pseudo-fonction

```
#define COMPOSE(x,y) x ## _ ## y (x, y)
```

génère la réécriture du motif

```
x = COMPOSE(rouge,vert)
```

en

```
x =  rouge_vert (rouge, vert)
```

On remarquera qu'une espace est insérée avant et après le mot résultant de la concaténation.

5.1.5 Récursivité des définitions

Le mécanisme de réécriture mis en œuvre par le préprocesseur est récursif. Après chaque réécriture, ce dernier examine la chaîne obtenue pour y rechercher d'éventuelles nouvelles substitutions. Voici un exemple de définitions récursives.

```
recdef.c
#define NOMBRE_MAX    256
#define TAILLE_MAX    (NOMBRE_MAX*sizeof(int))

#define RANG(n)       ((float)(n) / (float)TAILLE_MAX)


RANG(i+1);
recdef.c
```

On peut vérifier en visualisant le résultat des macro-expansions que toutes les substitutions ont bien été effectuées.

```
$ gcc -E recdef.c
# 1 "recdef.c"
      ((float) (i+1) / (float)    (   256*sizeof(int)));
$
```

Les définitions circulaires, comme par exemple

```
#define MAX    TBUF-1
#define TBUF   MAX+1
```

produisent en général un message d'erreur.

5.2 Inclusion de fichiers

5.2.1 La directive include

La directive **include** permet d'importer dans un fichier le contenu d'un autre fichier. Ce mécanisme est en général réservé à l'inclusion de fichiers de déclarations, appelés **fichiers en-tête**, et traditionnellement suffixés par `.h` (pour *header*). Un fichier en-tête regroupe les déclarations de variables globales, les définitions de types, de pseudo-constantes, et de pseudo-fonctions communes à plusieurs fichiers. La syntaxe d'une inclusion de fichier est de l'une des deux formes suivantes :

directive-inclusion : `#include "`*fichier*`"`
`#include <`*fichier*`>`

Lorsque le préprocesseur rencontre une telle directive dans un fichier, il la remplace par le contenu du fichier spécifié. Lorsque le nom du fichier est écrit entre guillemets, le fichier est recherché dans le répertoire courant. L'écriture entre chevrons fait référence à des fichiers en-tête standard. Nous reviendrons sur ce point à la fin de cette section.

L'option `-I` permet de spécifier au préprocesseur un chemin de répertoire dans lequel rechercher les fichiers à inclure. Par exemple, la commande

```
gcc -g src.c -c -I$HOME/include
```

lance la compilation du fichier `src.c` en spécifiant que les fichiers mentionnés dans une directive de la forme

`#include "`*fichier*`"`

doivent être également recherchés dans le sous-répertoire `include` du répertoire d'accueil de l'utilisateur. On pourra également définir la variable `CFLAGS`, au niveau du shell ou d'un fichier `Makefile` (voir page 14), pour spécifier automatiquement les chemins de répertoires d'inclusion.

Le mécanisme d'inclusion de fichiers est récursif. Voici par exemple un fichier `inc_ab.c` incluant un fichier `inc_a.h`, lui-même incluant un fichier `inc_b.h`.

```
==== inc_ab.c ====
#include "inc_a.h"

char *ab = "Contenu du fichier inc-ab.c";
==== inc_ab.c ====
```

```
==== inc_a.h ====
#include "inc_b.h"

char *a = "Contenu du fichier inc-a.h";

==== inc_a.h ====
```

```
==== inc_b.h ====
char *b = "Contenu du fichier inc-b.h";
==== inc_b.h ====
```

Voici le fichier construit par le préprocesseur à partir du fichier `inc_ab.c` :

```
$ gcc -E inc_ab.c
# 1 "inc_ab.c"
# 1 "inc_a.h" 1
# 1 "inc_b.h" 1
char *b = "Contenu du fichier inc_b.h";
# 1 "inc_a.h" 2
```

```
char *a = "Contenu du fichier inc_a.h";

# 1 "inc_ab.c" 2


char *ab = "Contenu du fichier inc_ab.c";
$
```

On comprend mieux, sur cet exemple, pourquoi le préprocesseur génère dans le fichier qu'il construit des indications concernant les messages d'erreurs. Ces indications sont indispensables au compilateur pour donner une localisation correcte des erreurs de syntaxe (nom du fichier source et numéro de la ligne).

5.2.2 Fichiers en-tête

Il est justifié de définir un fichier en-tête chaque fois qu'une portion de code C est commune à plusieurs fichiers. Il peut alors être inclus dans chacun des fichiers concernés. Ainsi, une modification de cette portion de code commune est effectuée une seule fois, dans le fichier partagé. Il suffit ensuite de recompiler les différents fichiers incluant le fichier modifié, ce que la commande `make` effectue automatiquement si le fichier `Makefile` est correctement défini.

Une question importante qu'on peut se poser à ce sujet est : **quelles portions de code est-il légitime de partager entre plusieurs fichiers?**

Nous reviendrons très largement sur ce point dans le chapitre 8. On peut cependant déjà énoncer quelques principes de base. Dans la mesure où le code partagé est présent dans plusieurs fichiers, il ne doit pas contenir de définition d'objet (variables ou fonctions). Par conséquent, on placera seulement dans un fichier en-tête

- des directives au préprocesseur : inclusions de fichiers, définitions de pseudo-constantes et pseudo-fonctions...
- des déclarations consistant en
  - des définitions de type (voir 2.5);
  - des définitions de constante;
  - des prototypes de fonctions;
  - des références à des objets (voir 2.6.6).

Afin de permettre au compilateur de contrôler la correction des appels de fonction, on appliquera le principe suivant :

🛇 *Pour chaque fichier ⟨nom⟩*`.c`*, on définira un fichier en-tête ⟨nom⟩*`.h` *contenant les prototypes de toutes les fonctions globales définies dans ⟨nom⟩*`.c`*. Le fichier ⟨nom⟩*`.h` *devra être inclus dans le fichier ⟨nom⟩*`.c` *et dans tout fichier utilisant une de ces fonctions.*

5.2.3 Fichiers en-tête standard

La norme ANSI définit quinze fichiers en-tête devant faire partie de tout environnement C standard. Ils contiennent trois sortes de définition :

- les définitions des prototypes des fonctions de la bibliothèque standard;
- des fonctionnalités de la bibliothèque standard implémentées sous la forme de pseudo-fonctions;
- des définitions de pseudo-constantes définissant les caractéristiques propres à cette implémentation.

Voici la liste de ces fichiers :

assert.h : définition de la pseudo-fonction `assert` permettant de placer des assertions dans les programmes – voir 7.4.

ctype.h : définitions de pseudo-fonctions testant le type d'un caractère : lettre, chiffre, séparateur... – voir 7.2.1.

errno.h : codification des erreurs retournées par certaines fonctions de bibliothèques (sous UNIX, ce fichier sert également à décrire les erreurs retournées par les appels système – voir chapitre 9.2).

float.h : définitions de pseudo-constantes décrivant les limites de taille relatives au codage des nombres réels : exposant, partie entière, partie décimale, etc. Par exemple, la constante `DBL_EPSILON` définit la précision minimale d'un réel dans l'implémentation courante.

limits.h : définitions de pseudo-constantes décrivant les limites de taille des types `char`, `int`, `short`, `long`, et de leurs variantes signées ou sans signe. Par exemple, les constantes `LONG_MIN` et `LONG_MAX` définissent la valeur minimale et la valeur maximale d'un entier long pour l'implémentation courante.

locale.h : définitions relatives aux particularités des différentes implémentations internationales, comme :
  - les spécificités de l'alphabet : ordre lexicographique, correspondances minuscules/majuscules...
  - le formatage des nombres : virgule ou point décimal, règles de placement de signe...
  - le formatage des quantités monétaires;
  - les conventions d'écriture de la date et de l'heure;
  - certaines informations sur le langage;
  - etc.

  Les fonctions `setlocale` et `localeconv` permettent d'exploiter ces informations.

math.h : interface avec les fonctions mathématiques de la bibliothèque standard – voir page 31.

setjmp.h : déclarations des types et mécanismes implémentant la sauvegarde et restauration du contexte d'exécution d'un programme – voir 7.9.

signal.h : mécanisme standard de récupération des événements externes comme la prise en compte des interruptions ou des erreurs. Ce mécanisme reprend le mécanisme de signaux UNIX (voir 12.2.1). Les deux fonctions de la bibliothèque associées à ce mécanisme sont `signal` et `raise`.

stdarg.h : définitions de macros pour l'écriture de fonctions recevant un nombre variable de paramètres – voir 7.7.

stddef.h : définitions de types diverses : `ptrdiff_t` (différence de deux pointeurs), `size_t` (expression construite avec `sizeof`), `wchar_t` (caractères étendus), et de définition de la pseudo-constante `NULL`– voir 2.1.3.

stdio.h : types, pseudo-constantes et pseudo-fonctions relatives aux entrées-sorties – voir 10.2.1.

stdlib.h : prototypes des fonctions d'intérêt général de la bibliothèque standard : conversions de chaînes numériques (7.2.2), allocation dynamique (7.3), générateurs aléatoires, interaction avec le système (7.5, 7.8), etc.

string.h : interface avec les fonctions de manipulation de chaînes de caractères – voir 7.2.3.

time.h : interface avec les fonctions de manipulation du temps : date, heure, jour, mois... – voir 7.6.

La liste des fonctions définies dans chaque fichier en-tête est fournie dans l'annexe B.

On trouve également, de façon non standard, des fichiers en-tête propres au système, comme par exemple sous UNIX :

a.out.h et **nlist.h** : description d'un fichier exécutable – voir 12.1.2.

termcap.h et **curses.h** : définitions nécessaires à l'utilisation des bibliothèques de *fonctions vidéo*.

fcntl.h : définitions concernant les entrées-sorties de bas-niveau – voir 10.2.1 et 7.1.1;

utmp.h : description du fichier des utilisateurs connectés – voir 11.4.

L'emplacement des fichiers en-tête peut varier selon les systèmes et les compilateurs. Sous UNIX, les fichiers en-tête sont rangés par défaut dans le répertoire `/usr/include`. Il est donc possible de les inclure par une directive de la forme :

```
#include "/usr/include/⟨fichier⟩"
```

Cependant, cette forme est à éviter car un programme ainsi écrit n'est pas portable. De plus, si on utilise un compilateur autre que le compilateur original, comme le compilateur gcc par exemple, le répertoire des fichiers en-tête standard peut être différent. La façon normale de spécifier un fichier en-tête de l'environnement consiste à le placer entre chevrons (caractères < et >). Par exemple, on fait référence au fichier `ctype.h` par

```
#include <ctype.h>
```

5.3 Compilation conditionnelle

Les directives de compilation conditionnelle sont **if**, **ifdef**, **ifndef**, **elif**, **else** et **endif**. Elles permettent de tester l'existence et la valeur de pseudo-constantes; en fonction du résultat de ce test, le préprocesseur conserve ou supprime certaines lignes du fichier source.

5.3.1 Test d'existence d'une pseudo-constante

La syntaxe générale du test d'existence d'une pseudo-constante est

directive-conditionnelle : **#ifdef** *identificateur*
partie-alors
[**#else**
partie-sinon]
#endif

La non-existence d'une pseudo-constante peut être également testée :

directive-conditionnelle : **#ifndef** *identificateur*
partie-alors
[**#else**
partie-sinon]
#endif

La sémantique de la directive `ifdef` (resp. `ifndef`) est la suivante : si la pseudo-constante testée existe (resp. n'existe pas), le préprocesseur recopie les lignes de la *partie alors* dans le fichier généré pour le compilateur. Dans le cas contraire, et si la directive `else` est présente, le préprocesseur recopie les lignes de la *partie sinon*. La directive `elif`, signifiant *sinon si*, permet d'enchaîner plusieurs tests imbriqués. Elle se place entre la directive `if` et la directive `else`, et il est possible d'en enchaîner plusieurs à la suite.

L'exemple suivant montre comment tester l'existence d'une pseudo-constante avant de la définir, afin d'éviter une double définition :

```
#ifndef LOCALBIN
# define LOCALBIN    "/usr/local/bin"
#endif /* LOCALBIN */
```

Il est parfois utile de tester l'existence d'une pseudo-constante pour en annuler la définition :

```
#ifdef getchar
# undef getchar
#endif /* getchar */
#define getchar _getchar_
```

Une application importante de l'inclusion conditionnelle consiste à prévenir les inclusions multiples d'un même fichier en-tête. On associe pour cela une pseudo-constante à chaque nom de fichier en-tête dont on teste la définition au début du fichier. Dans les exemples présentés dans cet ouvrage, un fichier de nom ⟨*nom*⟩`.h` est associé à la pseudo-constante ⟨*NOM*⟩`_H`. Il commence par les lignes

```
#ifndef ⟨NOM⟩_H
#define ⟨NOM⟩_H
```

et se termine par

```
#endif /* ⟨NOM⟩_H */
```

De cette façon, son contenu est recopié au plus une fois dans chaque fichier l'utilisant.

Une autre application de la compilation conditionnelle est le paramétrage des différentes versions d'un même logiciel. Dans l'exemple suivant, l'inclusion d'un fichier de définitions est conditionnée par l'existence d'un symbole

correspondant au type de la machine cible.

```
#ifdef IRIS3XXX
# include "iris3xxx.h"
#endif /* IRIS3XXX */

#ifdef IRIS4D
# include "iris4d.h"
#endif /* IRIS4D */
```

On trouve encore relativement souvent dans les programmes des constructions destinées à tracer son exécution lors de la mise au point (voir à ce sujet la remarque 20 page 166) :

```
#ifdef DEBUG
 printf("indice_minimun = %d\n", indice_minimum);
#endif /* DEBUG */
```

Sur cet exemple, la recopie de l'instruction est conditionnée par l'existence de la pseudo-constante DEBUG : si le symbole DEBUG est défini lorsque le préprocesseur lit ces trois lignes, elles sont remplacées par

```
printf("indice_minimum = %d\n", indice_minimum);
```

et elles sont supprimées sinon. Ce mécanisme est généralisable de la façon suivante :

```
#ifdef DEBUG
# define TRACE(s) printf s
#else
# define TRACE(ignore)
#endif /* DEBUG */
```

Il est maintenant possible décrire :

```
TRACE(("Appel de f()"))
TRACE(("Appel de f(%d,%d)", arg1, arg2))
```

Si la variable DEBUG est définie, ces expressions seront expansées en :

```
printf("Appel de f()")
printf("Appel de f(%d,%d)", arg1, arg2)
```

5.3.2 Prise en compte de l'implémentation du compilateur

Un programme peut être paramétré par l'implémentation du compilateur. Nous avons vu page 153 qu'avec une implémentation ANSI, la variable __STDC__ est prédéfinie avec une valeur non nulle. Cette pseudo-constante peut être testée afin de conditionner l'utilisation de spécificités ANSI. Considérons par exemple la définition du fichier debug.h suivante :

```
debug.h
#ifndef DEBUG_H
#define DEBUG_H

#ifdef DEBUG
```

```
# ifdef __STDC__
#  define IDEBUG(x)  printf("\"%s\":%d: %s vaut %d\n",\
                              __FILE__, __LINE__, #x, x);
# else
#  define IDEBUG(x)  printf("\"%s\":%d: %d\n",\
                              __FILE__, __LINE__, x);
# endif /* __STDC__ */

#else
# define IDEBUG(ignore)
#endif /* DEBUG */

#endif  /* DEBUG_H */
```
debug.h

La définition de la pseudo-fonctions IDEBUG est conditionnée par l'existence des pseudo-constantes DEBUG et __STDC__. Dans le cas où le compilateur supporte la norme ANSI, la définition des trois pseudo-fonctions utilise le mécanisme de génération de chaîne de caractères présenté page 158.

La session suivante montre trois compilations différentes d'un fichier de test, le fichier debug.c. Les deux premières sont effectuées avec le compilateur **gcc** qui supporte les fonctionnalités ANSI; la pseudo-constante DEBUG est définie seulement lors des deux dernières compilations.

```
$ cat -n debug.c
     1  #include "debug.h"
     2
     3  main()
     4  {
     5      int i = 3;
     6
     7      IDEBUG(i);
     8      IDEBUG(i==0);
     9  }
$
$ gcc -g debug.c -o debug
$ debug
$
$ gcc -g -DDEBUG debug.c -o debug
$ debug
"debug.c":7: i vaut 3
"debug.c":8: i==0 vaut 0
$
$ cc -DDEBUG debug.c -o debug
$ debug
"debug.c":7: 3
"debug.c":8: 0
$
```

Remarque 20 *Le fait de parsemer le code d'un programme de directives de débogage nuit à sa lisibilité. D'autre part, cette technique est sans intérêt lorsqu'il est possible d'utiliser des débogueurs symboliques comme le débogeur* **gdb**

(voir page 230). Elle doit être réservée à des cas très particuliers pour lesquels le débogueur est inopérant.

5.3.3 Évaluation de pseudo-expressions

Il est possible de former des expressions interprétables par le préprocesseur. Ces expressions sont des expressions constantes restreintes, construites au moyen :

- de constantes caractère ou entières,
- de parenthèses,
- des opérateurs unaires - ! et ~,
- des opérateurs binaires + - * / % & | << >> < <= > >= == != && ||,
- de l'opérateur conditionnel ?:,
- de conversions vers les types autorisés,
- de l'opérateur unaire `defined`.

L'opérateur `defined` s'applique sur une pseudo-constante; l'expression

`defined` ⟨*pseudo-constante*⟩

vaut un si ⟨*pseudo-constante*⟩ est définie et zéro sinon. Les autres opérateurs ont leur signification habituelle. Toute pseudo-constante définissant une expression constante restreinte peut également être réutilisée dans une construction.

L'évaluation d'une pseudo-expression est déclenchée par la directive `if`, dont la syntaxe est :

directive-conditionnelle : `#if` *pseudo-expression*
partie-alors
[`#else`
partie-sinon]
`#endif`

La *partie-alors* est recopiée si le résultat de l'évaluation est différent de zéro. Dans le cas contraire, si elle existe, la *partie-sinon* est recopiée.

Une première application, très simple mais très utile, est la mise en commentaire d'une région de programme. La solution triviale, consistant à placer un `/*` et un `*/` respectivement au début et à la fin de la région, ne marche pas si cette région contient elle-même des commentaires. En effet, le mécanisme de définition de commentaires n'étant pas algébrique, il n'est pas possible d'imbriquer plusieurs commentaires les uns dans les autres.

La bonne façon d'effectuer cela, consiste à placer au début de la région à commenter, la directive

```
#if 0
```

et à la fin, la directive

```
#endif /* 0 */
```

Le fichier suivant `cppif` illustre l'utilisation d'expressions constantes; il s'agit du calcul d'une constante `TAILLEMAX` utilisée pour définir la dimension d'un vecteur. Cette expression, définie à partir des deux pseudo-constantes `TAILLE` et `TBLOC`, calcule le plus petit multiple de `TBLOC` supérieur ou égal à `TAILLE`.

La constante TBLOC est la chaîne 256; la constante TAILLE est définie par défaut égale à TBLOC. L'expression TAILLE%TBLOC == 0 est évaluée par le préprocesseur après substitution de TAILLE et TBLOC par leurs valeurs respectives.

cppif.c

```
#define TBLOC 256

#if !defined TAILLE
#  define TAILLE TBLOC
#endif

#if TAILLE%TBLOC == 0
#  define TAILLEMAX TAILLE
#else
#  define TAILLEMAX ((TAILLE/TBLOC+1)*TBLOC)
#endif

static char buffer[TAILLEMAX];

main()
{
    printf("Taille du vecteur: %d caracteres\n",
            sizeof buffer);
}
```

cppif.c

Bien qu'effectuant une entrée-sortie, nous n'avons pas inclus le fichier en-tête <stdio.h> pour éviter de charger le résultat du prétraitement. Voici le résultat de deux passages du préprocesseur sur ce fichier, le premier sans définir la variable TAILLE, et le second en lui donnant la valeur 257.

```
$ gcc -E cppif.c
# 1 "cppif.c"

static char buffer[256   ];

main()
{
    printf("Taille du vecteur: %d caracteres\n",
            sizeof buffer);
}
$
$ gcc -E cppif.c -DTAILLE=257
# 1 "cppif.c"

static char buffer[((257 /256 +1)*256 ) ];

main()
{
    printf("Taille du vecteur: %d caracteres\n",
            sizeof buffer);
}
$
```

Seule la pseudo-expression utilisée dans la directive `if` est évaluée. La définition de `TAILLEMAX` est expansée de façon classique lors de son utilisation dans la déclaration du vecteur. Le calcul correspondant est effectué par le compilateur :

```
$ gcc -g cppif.c -DTAILLE=255 -o cppif
$ cppif
Taille du vecteur: 256 caracteres
$
$ gcc -g cppif.c -DTAILLE=256 -o cppif
$ cppif
Taille du vecteur: 256 caracteres
$
$ gcc -g cppif.c -DTAILLE=257 -o cppif
$ cppif
Taille du vecteur: 512 caracteres
$
```

5.4 Pilotage des messages d'erreur

Un message d'erreur du compilateur C contient trois informations :

- le nom du fichier contenant l'erreur;
- le numéro de la ligne de l'erreur;
- la nature de l'erreur.

Par exemple, avec le compilateur `gcc`, le message

```
erreur.c: In function main:
erreur.c:3: 'i' undeclared (first use this function)
erreur.c:3: (Each undeclared identifier is reported only once
erreur.c:3: for each function it appears in.)
```

ou encore, avec le compilateur traditionnel, le message

```
"erreur.c", line 3: i undefined
```

signalent une erreur sur la troisième ligne du fichier `erreur.c`. La directive `line` permet de modifier la localisation de l'erreur.

```
=========== line.c ===========
main()
{
    3 = 4;
    3 = 4;

# line 16 "c.enil"
    3 = 4;
    3 = 4;
}
=========== line.c ===========
```

Les deux premiers messages d'erreur lors de la compilation de `line.c` sont émis avec les identifications de fichier et de ligne normales. Les deux derniers sont modifiés par l'utilisation de la directive `line`.

```
$ gcc -g line.c
line.c: In function main:
line.c:3: invalid lvalue in assignment
line.c:4: invalid lvalue in assignment
c.enil:16: invalid lvalue in assignment
c.enil:17: invalid lvalue in assignment
$
```

L'intérêt de cette directive est de permettre une localisation correcte des erreurs de syntaxe dans le cas très particulier de la génération automatique de programmes C, à partir de portions de programmes et d'une spécification. Si les portions de programme contiennent des erreurs, celles-ci seront recopiées telles quelles dans le programme généré. Il est important, afin de faciliter la mise au point, que le compilateur C localise ces erreurs dans le fichier contenant les portions de programme originales écrites par le programmeur, et non dans le fichier généré par le générateur automatique. Pour cela, il suffit que le générateur automatique insère dans le fichier généré des directives `line` indiquant la localisation correcte des messages d'erreur. C'est de cette façon que procèdent, par exemple, les générateurs d'analyseur **LEX(1)** et **YACC(1)**.

5.5 Un exemple de construction générique

Nous allons terminer ce chapitre avec l'exemple d'un ensemble de pseudo-fonctions de manipulation de *pile*. Il illustre un autre aspect intéressant du préprocesseur, à savoir la possibilité de génériciser une construction.

Les opérations définies sur une pile sont classiquement : *empiler* un élément, *dépiler* le *sommet de pile*, consulter le *sommet de pile*, et tester si la pile est vide ou pleine. Nous allons définir une pile comme un couple $\prec base, prochain \succ$ où *base* est un vecteur et *prochain* un pointeur référençant le premier élément libre de la pile. Lorsque la pile est vide, il pointe sur le début du vecteur. Voici, sur ce modèle, la déclaration d'une pile d'entiers :

```
struct
{
    int base[TAILLE_PILE];
    int *prochain;
} pile_d_entiers;
```

Il est également possible de définir le type d'une pile par :

```
struct pile_d_entiers
{
    int base[TAILLE_PILE];
    int *prochain;
};
typedef struct pile_d_entiers pile_d_entiers;
```

Si pile_d_entiers est un objet de ce type, son initialisation est effectuée par l'instruction :

```
pile_d_entiers.prochain = pile_d_entiers.base;
```

qui place le pointeur prochain de la structure pile_d_entiers sur le fond de pile. L'empilement d'une valeur v se fait alors par :

```
*pile_d_entiers.prochain++ = v;
```

En effet, pour empiler un élément, on doit le recopier dans la prochaine case libre, puis incrémenter le pointeur référençant cette case. Le dépilement est l'opération symétrique, c'est-à-dire l'expression :

```
*--pile_d_entiers.prochain
```

Le *sommet de pile* est donné par l'expression

```
*(pile_d_entiers.prochain - 1)
```

et la condition *pile vide* s'écrit simplement

```
pile_d_entiers.prochain == pile_d_entiers.base
```

La condition *pile pleine* est un peu plus complexe, mais n'est pas non plus très difficile à écrire (voir le fichier pile.h présenté plus loin).

Un problème demeure cependant qui est le suivant : il est nécessaire d'écrire un ensemble de pseudo-fonctions pour chaque type de pile. Nous allons voir comment généraliser ces fonctions au moyen d'un ensemble générique de pseudo-fonctions. La déclaration de cette sorte de pile nécessite trois paramètres : son type, son nom, et sa taille. Elle peut être décrite au moyen de la pseudo-fonction suivante :

```
#define PILE_DECLARER(type, ident, taille) \
struct \
{\
    type base[taille+1];\
    type *prochain;\
} ident
```

On peut par exemple déclarer une pile put_buffer de 256 caractères par :

```
PILE_DECLARER(char, put_buffer, 256);
```

Cette pseudo-fonction est récrite par le préprocesseur en :

```
struct
{
    char base[256];
    char *prochain;
} put_buffer;
```

Il suffit maintenant de paramétrer les pseudo-fonctions de manipulation de pile avec le nom de la pile pour disposer d'un ensemble générique de pseudo-fonctions de manipulation de pile. Le fichier pile.h suivant contient la liste de ces définitions

```
==== pile.h ====
#ifndef PILE_H
#define PILE_H
```

```
#define PILE_DECLARER(type, ident, taille) \
    struct \
    {\
        type base [taille];\
        type *prochain;\
    } ident

#define PILE_DEFTYPE(type, ident, taille) \
    struct ident \
    {\
        type base [taille];\
        type *prochain;\
    };\
    typedef struct ident ident

#define PILE_TAILLE(ident)  \
    (sizeof(ident).base/sizeof(ident).base[0])

#define PILE_LIMITE(ident) \
    ((ident).base + PILE_TAILLE((ident)))

#define PILE_INIT(ident) \
    (ident).prochain = ((ident).base)

#define EMPILER(ident,e)     (*(ident).prochain++ = (e))
#define DEPILER(ident)       (*--(ident).prochain)

#define PILE_VIDE(ident) \
    ((ident).prochain == (ident).base)
#define PILE_PLEINE(ident) \
    ((ident).prochain >= PILE_LIMITE(ident))

#define PILE_SOMMET(ident) (*((ident).prochain-1))

#endif  /* PILE_H */
```
pile.h

Le programme suivant `tpile.c` est un exemple d'utilisation de ces définitions.

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

#include "pile.h"

PILE_DEFTYPE(int, Pile, 1024);

main()
{
    Pile pile;
    PILE_DECLARER(int, ptmp, 5);
    int i=1;

    PILE_INIT(pile);
    PILE_INIT(ptmp);
    while (!PILE_PLEINE(ptmp))
```

```
    {
        EMPILER(ptmp, i);
        i++;
    }
    printf("- Pile ptmp pleine:\n");
    printf("    sommet de ptmp = %d\n", PILE_SOMMET(ptmp));

    if (!PILE_PLEINE(pile) && !PILE_VIDE(ptmp))
    {
        EMPILER(pile, DEPILER(ptmp));
        printf("- EMPILER(pile, DEPILER(ptmp)):\n");
        printf("    sommet de ptmp = %d\n",
               PILE_SOMMET(ptmp));
        printf("    sommet de pile = %d\n",
               PILE_SOMMET(pile));
    }
}
```

tpile.c

La première instruction définit l'identificateur de type `Pile` comme le type *pile de 1024 entiers.* La fonction `main` déclare deux piles, `pile` et `ptmp`, les initialise, puis teste différents empilements et dépilements.

```
$ tpile
- Pile ptmp pleine:
    sommet de ptmp = 5
- EMPILER(pile, DEPILER(ptmp)):
    sommet de ptmp = 4
    sommet de pile = 5
$
```

La réécriture du fichier `tpile.c` par le préprocesseur produit le programme suivant (pour plus de lisibilité, nous avons effectué un formatage du programme récrit) :

tpile.c_E

```
struct   Pile
{
   int  base [1024];
   int  *prochain;
};
typedef struct Pile Pile  ;

main()
{
   Pile pile;
   struct
   {
      int  base [5];
      int  *prochain;
   } ptmp;
   int i=1;

   (pile).prochain = ((pile).base) ;
   (ptmp).prochain = ((ptmp).base) ;
```

```
    while(!((ptmp).prochain
    >=((ptmp).base
       +(sizeof((ptmp)).base/sizeof((ptmp)).base[0]))))
    {
       (*(ptmp).prochain++ = (i)) ;
       i++;
    }
    printf ("- Pile ptmp pleine:\n");
    printf ("    sommet de ptmp = %d\n",(*((ptmp).prochain-1)));

    if(! ((pile).prochain
         >=((pile).base
            + (sizeof ((pile)).base/sizeof((pile)).base[0])))
       && !((ptmp).prochain == (ptmp).base))
    {
       (*(pile).prochain++ = ((*-- (ptmp).prochain))) ;
       printf("- EMPILER (pile, DEPILER (ptmp)):\n");
       printf("    sommet de ptmp = %d\n", (*((ptmp).prochain-1)));
       printf("    sommet de pile = %d\n", (*((pile).prochain-1)));
    }
}
```

tpile.c_E

On remarquera, par exemple, que le calcul de l'adresse du dernier élément de la pile lors du test `PILE_PLEINE(tmp)`, c'est-à-dire l'expression

```
((ptmp).pile + (sizeof((ptmp)).pile/sizeof((ptmp)).pile[0]))
```

est complètement résolu par le compilateur. En effet, les expressions intervenant dans le calcul, c'est-à-dire

```
ptmp.pile
sizeof(ptmp.pile)
sizeof(ptmp.pile[0])
```

sont toutes des expressions constantes.

Remarque 21 *On pourra améliorer ce module en remplaçant la définition de la pile générique par*

```
struct ⟨ident⟩
{
    ⟨type⟩ *pile;
    ⟨type⟩ *sommet;
    int taille;
};
typedef struct ⟨ident⟩ ⟨ident⟩;
```

dans laquelle la mémoire de la pile est allouée dynamiquement lors de l'initialisation :

```
#define PILE_INIT(ident) \
    ((ident).pile = malloc(sizeof(*pile)*NBELEM_PAR_BLOC), \
    (ident).sommet = ((ident).pile), \
    (ident).taille = NBELEM_PAR_BLOC)
```

et agrandie lorsque la pile est pleine (ces mécanismes sont détaillés en 7.3).

Chapitre 6

Manipulation d'adresses

6.1 Expressions et pointeurs

6.1.1 Valeur d'un pointeur

Nous avons défini en 2.1.2 un *pointeur* comme une variable ou une constante, dont la valeur est une adresse. Il peut s'agir de l'adresse d'une variable (entier, caractère, réel, structure ou pointeur), d'un vecteur ou d'une fonction.

L'adresse et le type de l'objet pointé ne sont pas dissociables (voir par exemple la figure 2.1 page 52). Par conséquent, nous considèrerons que la valeur d'un pointeur vers un objet d'adresse α et de type t est un couple que nous noterons :

$$\prec \alpha, t \succ$$

Un pointeur peut être une variable ou une constante. Nous avons vu en 2.2 divers exemples de définitions de variables de type pointeur, et en 2.3 nous avons présenté en détail les mécanismes de construction correspondants. Une constante pointeur peut être un identificateur de vecteur ou de fonction, ou une expression de type pointeur différente d'une g-valeur.

La façon la plus simple de définir une constante pointeur est d'appliquer l'opérateur de référencement `&` sur une g-valeur. Il est également possible de transformer une expression en une constante pointeur par conversion de type. Par exemple, si ⟨*exp*⟩ est une expression de valeur α, l'expression

`(char *)` ⟨*exp*⟩

est une constante pointeur de valeur

$$\prec \alpha, \text{char} \succ$$

Les opérateurs acceptant, sous certaines conditions, des pointeurs comme opérandes, sont :

- l'affectation simple : `=`;
- les opérateurs additifs : `+`, `-`, `+=`; `-=`, `++`, `--`;
- les indirections : `*`, `->`;
- les conversions de type et le calcul de taille : (*type*), `sizeof`;

– les constructeurs de listes et d'expressions conditionnelles : `,`, `?:`;
– les opérateurs de comparaison : `<`, `<=`, `>`, `>=`, `==`, `!=`.

6.1.2 Affectations de pointeurs et conversions de types

Si $\langle v \rangle$ et $\langle e \rangle$ sont respectivement une variable et une expression de type pointeur, et si la valeur de $\langle e \rangle$ est $\prec \alpha, t \succ$, l'expression

$$\langle v \rangle \texttt{ = } \langle e \rangle$$

recopie l'adresse α dans la variable $\langle v \rangle$. En toute logique, une telle expression ne devrait être correcte que lorsque $\langle v \rangle$ est de type *pointeur sur un objet de type t*. Cependant, le compilateur tolère une affectation entre une expression et une variable pointant sur un objet de type différent. Il y a dans ce cas émission d'un message d'avertissement de la forme :

```
ill-ass.c:7: warning: assignment between incompatible
                      pointer types
```

Une affectation entre un pointeur et un objet de type caractère ou entier est également tolérée :

```
ill-ass.c:6: warning: assignment of pointer from integer
                      lacks a cast
```

Par contre, elle est considérée comme une erreur fatale si l'un des deux opérandes est un réel.

```
ill-ass.c:8: incompatible types in assignment
```

Afin de comprendre pourquoi le compilateur tolère ces combinaisons illégales de type, considérons l'expression `p = q` où `p` et `q` sont deux pointeurs de valeur respectivement $\prec \alpha_p, t_p \succ$ et $\prec \alpha_q, t_q \succ$. La valeur de `p` après l'effet de bord est $\prec \alpha_q, t_p \succ$, quels que soient les types t_p et t_q; seule l'adresse α_q est prise en compte pour l'évaluation de l'affectation.

Cela sous-entend qu'une adresse est toujours une adresse, que ce soit l'adresse d'un caractère, d'un entier, ou de tout autre objet. Cela sous-entend également que la valeur entière α et l'adresse α ont en machine le même codage, ce qui n'est, par contre, pas le cas si α est un réel.

En réalité, il arrive que le codage d'un pointeur diffère selon le type de l'objet pointé[1]. Un compilateur C traditionnel doit au minimum garantir que la conversion d'un pointeur quelconque vers un pointeur de caractères, et réciproquement d'un pointeur de caractères vers un pointeur quelconque, s'effectue sans perte d'information. En C ANSI, cette propriété est satisfaite par le type

`void *`

encore appelé **pointeur générique**. De plus, les conversions implicites entre un pointeur générique et n'importe quel autre pointeur sont légales et ne pro-

[1] Par exemple, les formats des pointeurs de données et des pointeurs de fonctions peuvent être différents.

duisent pas de message d'erreur. Par contre, les additions, soustractions, comparaisons et indirections sur des pointeurs génériques sont illégales.

Même lorsque les différents types de pointeur ont la même représentation machine, ce qui est généralement le cas, les messages d'avertissement du compilateur ne doivent pas être pris à la légère. L'expérience montre que la plupart du temps, ils signalent des erreurs ayant une incidence réelle sur le déroulement du programme. Par exemple, on a écrit :

```
for (p=src, q=dst; *p; *q++ = *p++)
    ;
*q = '\000';
```

dans le but de recopier la chaîne *src* à l'adresse *dst*. Une faute de frappe, et l'instruction devient :

```
for (p=src, q=dst; *p; q++ = *p++)
    ;
*q = '\000';
```

Le compilateur signale, sur la première ligne, une combinaison illégale entre un pointeur et un entier, message bien utile pour détecter l'étoile oubliée dans la dernière expression.

L'oubli d'une étoile est une faute de frappe relativement fréquente, que ce soit dans une expression ou dans une déclaration. L'écriture d'expressions complexes est également une importante source d'erreur. Il est donc fondamental de tenir compte de tous les messages d'erreur émis par le compilateur, même lorsqu'il s'agit seulement de messages d'avertissement. Cela permet de déceler immédiatement la plus grande partie des erreurs de manipulation de pointeurs. On appliquera par conséquent scrupuleusement la règle suivante :

Il faut toujours former des expressions homogènes, en utilisant au besoin des opérateurs de conversion explicite, et ne jamais tirer parti des tolérances du compilateur utilisé.

Les mêmes contraintes de type régissent les comparaisons de pointeurs. Par exemple, la comparaison entre un pointeur et un entier produit le message d'avertissement :

```
ill-comp.c:6: warning: comparison between pointer and integer
```

Le seul cas légal est la comparaison avec (ou l'affectation de) la constante `NULL` de valeur *zéro*. D'autre part, la comparaison de deux pointeurs est légale seulement si ces deux pointeurs pointent vers un même objet, par exemple vers deux éléments d'un même vecteur ou deux champs d'une même structure, avec une extension dans le cas d'un vecteur : l'adresse du premier élément situé après la fin du vecteur est comparable avec l'adresse de n'importe quel élément de ce vecteur. Par contre, une indirection sur cette adresse est illégale. On notera également que si `v` est un vecteur, le résultat de la comparaison de `v-1` avec l'adresse d'un élément du vecteur est indéterminé.

Si `p` et `q` sont deux pointeurs vérifiant les conditions que nous venons d'énoncer, l'expression

```
p < q
```

vaut 1 si l'élément pointé par `p` est situé avant l'élément pointé par `q` et 0 sinon. S'ils pointent vers deux objets différents, le résultat est indéterminé et dépend de l'implémentation.

On peut en déduire que les champs d'une structure sont rangés, par le compilateur, dans l'ordre dans lequel ils sont déclarés. De plus, l'adresse du premier champ est toujours égale à celle de la structure. Par contre, les champs ne sont pas nécessairement consécutifs, le compilateur pouvant procéder à des réalignements.

En résumé, on retiendra la règle suivante :

Deux pointeurs ne peuvent être comparés que s'il référencent un même objet. Dans le cas d'un vecteur v, l'adresse v + `sizeof` *v est comparable avec celle d'un élément quelconque de v.*

6.1.3 Additions et pointeurs

Il est possible d'utiliser des pointeurs comme opérandes d'opérateurs additifs. La liste des constructions syntaxiquement légales est donnée dans le tableau 6.1.

Opérateur	Opérande 1	Opérande 2	Résultat
`+`	*pointeur*	*entier*	*pointeur*
`+`	*entier*	*pointeur*	*pointeur*
`-`	*pointeur*	*entier*	*pointeur*
`-`	*pointeur*	*pointeur*	*entier*
`+=`	*pointeur*	*entier*	*pointeur*
`-=`	*pointeur*	*entier*	*pointeur*
`++`	*pointeur*	—	*pointeur*
`--`	*pointeur*	—	*pointeur*

Tableau 6.1: Opérateurs additifs et pointeurs

Le type de la différence de deux adresses est un type entier dont la taille dépend de l'implémentation. Il est prédéfini, sous le nom `ptrdiff_t`, dans le fichier en-tête `<stddef.h>` (voir 2.1.3).

Un entier, lorsqu'il est ajouté ou retranché à un pointeur, représente un nombre d'objets du type de l'objet pointé. Il est automatiquement converti en une valeur de décalage, c'est-à-dire un nombre d'octets. Ainsi,

$$\prec \alpha, \texttt{char} \succ + 1 \quad \text{vaut} \quad \prec \alpha + 1, \texttt{char} \succ$$

alors que

$$\prec \alpha, \texttt{long} \succ + 1 \quad \text{vaut} \quad \prec \alpha + 4, \texttt{long} \succ.$$

Plus généralement,

$$\begin{cases} (1) & \prec \alpha, t \succ + n & \text{vaut} & \prec \alpha + n * \texttt{sizeof}(t), t \succ \\ (2) & \prec \alpha, t \succ - n & \text{vaut} & \prec \alpha - n * \texttt{sizeof}(t), t \succ \\ (3) & \prec \alpha, t \succ - \prec \alpha', t \succ & \text{vaut} & (\alpha - \alpha')/\texttt{sizeof}(t) \end{cases}$$

Remarque 22 *On vérifiera que cette sémantique est cohérente, et qu'on a bien* $(p+1)-p=1$.

Remarque 23 *On déduit de* (3) *que la différence entre deux pointeurs de types différents n'a pas de sens.*

Chaque type de base peut avoir en machine un alignement imposé dépendant de sa taille en octets. Le seul type n'ayant aucune contrainte d'alignement est en principe le type `char`. Par contre, un entier long sur quatre octets doit généralement démarrer à une adresse multiple de deux ou quatre octets. Par conséquent, si `pl` est un pointeur d'entiers longs, il y a de fortes chances pour que l'évaluation de l'expression

```
*(long *)((char *)pl + 1)
```

qui incrémente l'adresse contenue dans `pl` de un octet, produise une erreur à l'exécution. Il est possible d'éviter ce genre de problèmes en respectant le principe suivant :

Dans une application standard, il ne faut pas effectuer d'opération sur un pointeur dont le type a été modifié.

La restriction énoncée page 177 concernant les comparaisons de pointeurs s'applique également ici : les adresses utilisées comme opérandes d'une expression et l'adresse résultant de son évaluation doivent appartenir à un même objet.

Si $\langle v \rangle$ est un vecteur d'objets de type quelconque, et si $\langle p \rangle$ est un pointeur référençant un objet de $\langle v \rangle$, l'expression $\langle p \rangle + 1$ est toujours l'adresse de l'objet suivant. Cela permet, quel que soit le type de $\langle v \rangle$, de le parcourir au moyen de l'instruction

```
for (⟨p⟩ = ⟨v⟩; ⟨p⟩ < ⟨v⟩ + NOMBRE_D_ELEMENTS(⟨v⟩); ⟨p⟩++)
    ...
```

La pseudo-fonction `NOMBRE_D_ELEMENTS`, calculant le nombre d'éléments d'un vecteur, est définie page 112.

Dans le corps de boucle, l'expression `*`$\langle p \rangle$ permet d'accéder à l'élément courant; l'expression $\langle p \rangle$`++` incrémente le contenu de $\langle p \rangle$ de la taille en octets d'un objet du vecteur, ce qui correspond bien au passage à l'objet suivant. Enfin, l'adresse

```
⟨v⟩ + NOMBRE_D_ELEMENTS(⟨v⟩)
```

est l'adresse du premier élément n'appartenant pas à $\langle v \rangle$. Rappelons que c'est la seule adresse située hors du vecteur légalement comparable avec l'adresse d'un de ses éléments.

Cette technique permet de travailler directement avec les adresses des objets du programme, et donc d'accroître son efficacité, sans pour autant nuire à sa lisibilité.

6.1.4 Initialisation de pointeurs

L'erreur de programmation classique qui consiste à utiliser une variable sans l'avoir préalablement initialisée est souvent commise par les programmeurs expérimentant les pointeurs en C. Par exemple, le programme suivant :

```
memfaute.c
main()
{
    short *pi;

    *pi = 999;
}
memfaute.c
```

provoque sous UNIX une terminaison anormale lors de son exécution :

```
$ memfaute
Segmentation fault (core dumped)
$
```

Avec un compilateur C initialisant les variables automatiques à zéro, la valeur du pointeur `pi` est $\prec 0,$ `short` $\succ$, et `*pi` est la zone de 2 octets débutant à l'adresse mémoire 0. Par conséquent, on vient d'essayer d'écrire la valeur 999 à l'adresse logique zéro de l'espace mémoire attribué au processus utilisateur. Cette adresse est en général inaccessible, précisément pour faciliter la détection d'une utilisation de pointeur non initialisé. Dans le cas où l'initialisation à zéro n'est pas faite, il y a de fortes chances que l'adresse pointée par le pointeur soit également une adresse illégale.

De manière générale, il ne faut pas confondre la déclaration d'un pointeur et son initialisation. L'initialisation d'un pointeur peut se faire au moyen de l'opérateur unaire préfixe `&` qui permet d'obtenir l'adresse d'une g-valeur. Par exemple, si on a déclaré

```
struct sommet s, *ps;
```

l'instruction

```
ps = &s;
```

fait pointer le pointeur `ps` sur la structure `s`; si `fils` est un champ de la structure `s`, on peut maintenant y accéder par

```
s.fils
```

ou par

```
ps->fils
```

Un cas très fréquent d'utilisation incorrecte de pointeur se produit lors de la manipulation des chaînes de caractères. On rencontre par exemple souvent l'erreur consistant à déclarer un pointeur

```
char *buffer;
```

puis à l'utiliser en paramètre d'un appel de fonction, comme l'adresse d'une zone mémoire dont se sert la fonction :

```
scanf("%s", buffer);
```

Si le pointeur `buffer` n'a pas été initialisé sur une zone mémoire valide, la chaîne lue par la fonction `scanf` sera recopiée n'importe où en mémoire, et le plus souvent à l'adresse 0 qui, comme on l'a vu, est généralement protégée en écriture. Cette erreur est fréquemment commise lors de l'utilisation des fonc-

tions de bibliothèque construisant des chaînes de caractères, comme la fonction `sprintf` de formatage en mémoire (voir 7.1.2), ou les fonctions manipulation de chaînes de caractères de la bibliothèque `string` (voir 7.2.3). Par exemple, on déclare un pointeur :

```
char *term;
```

puis on écrit :

```
strcpy(term, "/dev/tty");
```

qui produit sous UNIX à l'exécution le message classique :

```
Segmentation fault (core dumped)
```

En effet, la fonction `strcpy` recopie la chaîne `"/dev/tty"` à l'adresse pointée par `term`, qui n'est pas, dans ce cas, une adresse valide. Cette erreur, très courante chez les débutants en C, est vraisemblablement due à une mauvaise compréhension de la syntaxe utilisée dans le manuel UNIX pour décrire la mise en œuvre des fonctions de bibliothèque. Par exemple, le synopsys de la fonction `strcpy` dans la page de manuel STRING(3) est :

```
char *strcpy(s1,s2)
char *s1, *s2;
```

Cela ne signifie pas qu'il faut déclarer les paramètres effectifs de type *pointeur de caractères*, mais que c'est ainsi qu'ils sont déclarés dans la fonction `strcpy`.

Une solution simple, consiste dans ce cas, à déclarer `buffer` comme un vecteur de caractères :

```
char buffer[TMAXBUF];
```

Mai il se pose alors le problème de fixer à l'avance une taille *"suffisamment grande"* pour la variable `buffer`. Cette façon de faire crée sur les logiciels des contraintes qu'il est de plus en plus difficile d'accepter. D'autre part, les modules écrits avec des limitations de taille à priori ne peuvent pas prétendre être réutilisables.

6.2 Pointeurs et chaînes de caractères

Les chaînes de caractères ne constituent pas un type à part entière du langage C. On dit encore que ce ne sont pas des objets de première catégorie. En particulier, il n'est pas possible de recopier une chaîne ou de comparer deux chaînes, au moyen d'opérateurs d'expressions.

Nous savons qu'une chaîne de caractères, en C, est une suite de caractères terminée par le caractère `\0`. Ainsi, `\"chaine\007"` est une constante ayant pour valeur l'adresse d'une zone mémoire dont les huit premiers octets contiennent les caractères `c`, `h`, `a`, `i`, `n`, `e`, 🔔 et `\0`.

La suite d'instructions

```
char *p = "ABCD";
char  c = *(p+2);
```

déclare un pointeur de caractères `p`, met dans `p` l'adresse du début de la chaîne `"ABCD"` (le pointeur `p` pointe alors sur le caractère `'A'`), et déclare une variable

caractère c dans lequel est recopié le troisième caractère de la chaîne, c'est-à-dire le caractère 'C'. Cela constitue une première façon de construire des chaînes de caractères : le compilateur initialise une zone mémoire avec la suite des caractères placés entre les guillemets et rajoute à la fin de cette zone le caractère \0.

Une chaîne de caractères définie au moyen de guillemets est une constante; par conséquent, son contenu ne peut être modifié.

Un vecteur de caractères peut aussi être rempli *à la main*; le dernier caractère doit être le caractère \0. Par exemple, la suite d'instructions

```
char ch[8];
ch[0] = 'a';
ch[1] = 'b';
ch[2] = '\0';
```

construit la chaîne représentée sur la figure 6.1.a. Cette chaîne est maintenant utilisable par toute fonction manipulant des chaînes de caractères, et en particulier par les fonctions standard de la bibliothèque **STRING (3)**, comme :

- la fonction de comparaison strcmp,
- la fonction de concaténation strcat,
- la fonction calculant la longueur strlen.

Ces fonctions sont détaillées en 7.2.3.

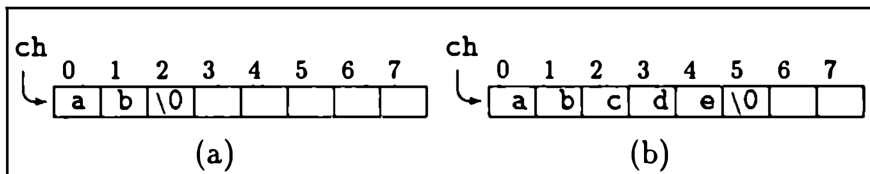


Figure 6.1: Exemples de chaînes de caractères

Par exemple l'exécution de

```
strcat(ch, "cde");
```

modifie la zone mémoire pointée par ch, en concaténant à la suite du caractère b les quatre caractères c, d, e, et \0. Le résultat de cette opération est représenté sur la figure 6.1.b; la valeur retournée par l'expression

```
strlen(ch)
```

est alors 5. Le fichier chaines.c illustre les différentes façons de manipuler les chaînes de caractères que nous venons d'énumérer.

```
chaines.c
#define TMAX    32

main()
{
    char alphabet_1[TMAX];
    char alphabet_2[TMAX];
    char *p;
```

```
    int  i;

    /*
     1) Construction de la chaine "ABCD...YZ"
        dans le vecteur 'alphabet_1'
     */
    for (i = 0; i < 26; i++)
        alphabet_1[i] = 'A' + i;
    alphabet_1[26] = '\0';
    printf("<1> Chaine 1 : %s\n", alphabet_1);

    /*
     2) Recopie de 'alphabet_1' dans
        'alphabet_2' avec STRCPY(3)
     */
    strcpy(alphabet_2, alphabet_1);
    printf("<2> Chaine 2 : %s\n", alphabet_2);

    /*
     3) La premiere occurrence caractere NULL
        marque la fin de chaine
     */
    alphabet_2[16] = '\0';
    printf("<3> Chaine 2 : %s\n", alphabet_2);

    /*
     4) Si p est de type (char*), les expressions
        p[i] et *(p+i) ont la meme valeur
     */
    p = alphabet_1 + 5;
    printf("<4> 6-eme caractere de chaine 1: %c\n", *p);

   /*
    5) Parcours d'une chaine de caracteres
    */
    printf("<5> Chaine 1 : ");
    for (p = alphabet_1; *p; p++)
        putchar(*p);
    printf("\n");
}
```

chaines.c

Voici le résultat de son exécution :

```
$ chaines
<1> Chaine 1 : ABCDEFGHIJKLMNOPQRSTUVWXYZ
<2> Chaine 2 : ABCDEFGHIJKLMNOPQRSTUVWXYZ
<3> Chaine 2 : ABCDEFGHIJKLMNOP
<4> 6-eme caractere de chaine 1: F
<5> Chaine 1 : ABCDEFGHIJKLMNOPQRSTUVWXYZ
$
```

6.3 Pointeurs et vecteurs

6.3.1 Similitudes et différences

Si p et v sont respectivement un pointeur et un vecteur d'objets de même type de base, l'expression

```
p = v
```

place dans p l'adresse du début du vecteur v. Dans ce cas, les expressions

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

sont toutes des expressions correctes ayant la même valeur. La seule différence entre les objets p et v réside dans le fait que p est une variable alors que v est une constante.

Par contre, lorsqu'on effectue des constructions plus complexes (vecteurs de vecteurs, vecteurs de pointeurs, pointeurs de pointeurs...), pointeurs et vecteurs ne sont plus interchangeables. Par exemple, s'il est possible de déclarer un paramètre de fonction ou un objet externe par

```
char *v[];
```

il n'est par contre pas légal d'écrire

```
char m[][];
```

En effet, on peut vérifier que dans ce dernier cas l'expression m[i] est indéfinie : l'expression m[i] est équivalente à *(m+i), or l'expression m + i ne peut être évaluée puisque m est un pointeur vers un vecteur de taille indéterminée. La déclaration

```
char m[][10];
```

est par contre correcte car m est un pointeur vers un vecteur de 10 caractères. Par conséquent, les deux expressions *(m+i) et m[i] sont définies et ont la même valeur, qui est le début du i$^{\text{ème}}$ vecteur de 10 caractères.

Par contre, un vecteur $\langle v_1 \rangle$ de vecteurs de $\langle n \rangle$ caractères et un vecteur $\langle v_2 \rangle$ de pointeurs de caractères

char $\langle v_1 \rangle$[][$\langle n \rangle$];
char *$\langle v_2 \rangle$[];

sont des objets différents. En effet, supposons que les deux objets référencent la même adresse α; la valeur de $\langle v_1 \rangle$ est $\prec \alpha,$ char [$\langle n \rangle$] $\succ$ alors que celle de $\langle v_2 \rangle$ est $\prec \alpha,$ char* $\succ$. Par conséquent les deux expressions $\langle v_1 \rangle$[i] et $\langle v_2 \rangle$[i] ont des valeurs différentes lorsque la valeur i est différente de zéro.

L'exemple le plus fréquemment rencontré est celui du vecteur de pointeurs de caractères qui est équivalent à un pointeur de pointeurs de caractères. En effet, avec les hypothèses précédentes (objets $\langle v_1 \rangle$ et $\langle v_2 \rangle$ référençant la même adresse α), les deux objets ont la même valeur $\prec \alpha,$ char* $\succ$. C'est la façon standard de coder un vecteur de chaînes de caractères, comme par exemple le paramètre argv de la fonction main sur lequel nous allons revenir dans la section suivante.

6.3.2 Vecteur de pointeurs et pointeur de pointeurs

Rappelons que la fonction main reçoit comme premier paramètre le nombre d'arguments de la commande et comme second paramètre la liste de ces arguments, codée comme un vecteur de chaînes de caractères (voir figure 1.2, page 24). Une façon simple de parcourir les arguments d'une commande a déjà été présentée dans le premier chapitre :

```
...
for (i=1; i<argc; i++)
    traiter(argv[i]);
...
```

Dans cette instruction, l'objet argv est utilisé comme un vecteur de pointeurs de caractères. Une seconde manière de procéder consiste à gérer argv comme un pointeur de pointeurs de caractères, comme c'est le cas dans le fichier argv_pp.

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

void traiter(char *);

main(int argc, char **argv)
{
    char *p;

    while (++argv, --argc)
        traiter(*argv);
}

void traiter(char *s)
{
    printf("    recu : %s\n", s);
}
argv_pp.c
```

Voici un exemple d'exécution de ce programme de test :

```
$ argv_pp -i -r f1 f2
    recu : -i
    recu : -r
    recu : f1
    recu : f2
$
```

6.4 Pointeurs et fonctions

6.4.1 Fonctions retournant un pointeur

Une fonction peut retourner un pointeur; il convient alors de prendre certaines précautions. En effet, une variable déclarée dans une fonction est par défaut

automatique, c'est-à-dire implantée dans la pile d'exécution (voir 2.6.2). Cela signifie en particulier que sa place est allouée à l'entrée de la fonction, et restituée à la sortie.

Il faut donc éviter l'erreur commise dans le programme suivant, consistant à retourner l'adresse d'une variable dynamique.

```
═══════ erreur_auto.c ═══════════════════════════════
#include <stdio.h>

char *ini_car();
void f_car();

main()
{
    char *p;

    p = ini_car();
    printf("Avant le second appel <%c>\n", *p);
    f_car();
    printf("Apres le second appel <%c>\n", *p);
}

char *ini_car()
{
    char c;

    c = '@';
    return(&c);
}

void f_car()
{
    char car;

    car = '#';
}
═══════════════════════════════ erreur_auto.c ═══════
```

A première vue, l'exécution de la fonction `f_car` ne peut pas provoquer d'effet de bord sur le caractère `*p` de la fonction `main`. Pourtant, lors de l'exécution du programme, l'appel de `f_car` modifie le contenu de l'octet pointé par le pointeur `p` de la fonction `main`.

```
$ erreur_auto
Avant le second appel <@>
Apres le second appel <#>
$
```

L'allocation/libération des variables automatiques obéissant à un mécanisme de pile, la variable `c` de la fonction `ini_car` et la variable `car` de la fonction `f_car` partagent la même adresse mémoire dans la pile d'exécution.

☹ *Lorsqu'une fonction retourne l'adresse d'une variable locale, celle-ci doit obligatoirement être une variable permanente, c'est-à-dire déclarée au moyen du mot-clé* `static`.

6.4.2 Passage par référence

Les paramètres de fonctions sont toujours passés par valeur; les modifications de paramètres dans la fonction appelée ne sont pas répercutées dans la fonction appelante. Un passage par référence doit donc être géré explicitement au moyen des opérateurs de manipulation d'adresse. Considérons le cas de figure suivant : on veut calculer le milieu `m` des points `p` et `q` de type

```
struct point
{
    float x;
    float y;
};
typedef struct point point;
```

On définit la fonction `milieu` de la façon suivante :

```
void milieu(point p1, point p2, point *p)
{
    p->x = (p1.x + p2.x) / 2;
    p->y = (p1.y + p2.y) / 2;
}
```

Lors de l'appel à la fonction `milieu`, le troisième paramètre est passé par référence : c'est l'adresse de la variable `m` et non sa valeur qui est transmise. Seule une g-valeur peut être passée par référence.

```
milieu(p, q, &m);
```

De cette façon, la fonction `milieu` effectue un effet de bord sur la structure `m` de la fonction appelante.

6.4.3 Fonctions passées en paramètres

Nous avons vu que le langage C permet de définir des pointeurs vers n'importe quelle construction, y compris vers des fonctions. Une application des pointeurs de fonctions est le passage de fonctions en paramètres d'un appel de fonction.

Pour utiliser une fonction $\langle g \rangle$ comme paramètre d'une fonction $\langle f \rangle$, il suffit de passer à $\langle f \rangle$ un pointeur vers $\langle g \rangle$. L'exemple suivant illustre ce mécanisme.

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

static void g(void (*)());
static void f1();
static void f2();

main()
{
    printf("\nTest de passage de fonction en parametre\n");
```

```
    g(f1);
    g(f2);
}

/*
 * Recoit en parametre un pointeur vers la fonction a appeler
 */
static void g(void(*f)())
{
    printf("  - g appelle ");
    f();
    printf("\n");
}

static void f1()
{
    printf("f1");
}

static void f2()
{
    printf("f2");
}
```
fnct_par.c

La fonction `main` du programme `fnct_par.c` appelle une fonction `g` avec comme paramètre, une première fois, une fonction `f1`, et une seconde fois, une fonction `f2`. La fonction `g` appelle la fonction dont elle reçoit l'adresse en paramètre.

```
$ fnct_par

Test de passage de fonction en parametre
  - g appelle f1
  - g appelle f2
$
```

On trouvera dans cet ouvrage plusieurs programmes comportant des passages de fonction en paramètres (voir par exemple la fonction `liste_itere` du fichier `cellule.c`, page 192).

6.4.4 Tables d'indirection de fonctions

La constitution d'une table d'indirection de fonctions est une autre application de l'utilisation de pointeurs de fonctions. Il s'agit d'un vecteur dont chaque élément est initialisé avec l'adresse d'une fonction. Une fois la table initialisée, il est possible d'appeler directement la i$^{\text{ème}}$ fonction de la table.

L'exemple suivant, dans lequel est défini un vecteur `tbf` de pointeurs de fonctions initialisé avec trois fonctions, respectivement `f0`, `f1`, et `f2`, illustre

cette technique. On peut former les objets suivants :

`tbf`	:	adresse d'un vecteur de pointeurs de fonctions;
`tbf[i]`	:	pointeur vers la i+1$^{\text{ème}}$ fonction;
`tbf[i](p)` `(*tbf[i])(p)`	:	appel de la i+1$^{\text{ème}}$ fonction avec le paramètre p.

Le programme est une simple boucle parcourant la table, et appelant, pour chaque élément la fonction qu'il référence, avec l'indice courant en paramètre.

```
═════════ ptfonc.c ═══════════════════════════
#include <stdio.h>

void f0(int), f1(int), f2(int);
void (*tbf[])(int) = { f0, f1, f2 };

main()
{
    int i;

    for (i=0; i<3; i++)
        tbf[i](i);
}

void f0(int i)
{
    printf("  f0 recoit %d\n", i);
}

void f1(int i)
{
    printf("  f1 recoit %d\n", i);
}

void f2(int i)
{
    printf("  f2 recoit %d\n", i);
}
═══════════════════════════════ ptfonc.c ═════
```

À chaque tour de boucle, une nouvelle fonction est appelée :

```
$ ptfonc
  f0 recoit 0
  f1 recoit 1
  f2 recoit 2
$
```

Une application possible des vecteurs de fonctions est l'association de *clés* (c'est-à-dire de caractères de contrôle entrés au clavier) à des fonctions. La suite de déclarations de l'exemple suivant associe les fonctions `lire`, `ecrire`, `menu`, `suivant`, `precedent` et `fin` aux caractères de contrôle `C-l`, `C-e`, `C-m`, `C-s`, `C-p` et `C-f`. Toute autre clé est associée à une fonction d'erreur.

```
cles.c
#include "cles.h"

extern void ecrire(void);
extern void erreur(void);
extern void fin(void);
extern void lire(void);
extern void menu(void);
extern void precedent(void);
extern void suivant(void);

void (* cle_fonct[])() =
{
   erreur, erreur, erreur, erreur, erreur, ecrire, fin,    erreur,
   erreur, erreur, erreur, erreur, lire,   menu,   erreur, erreur,
   erreur, erreur, erreur, suivant,erreur, erreur, erreur, erreur,
   erreur, erreur, erreur, erreur, erreur, erreur, erreur, erreur
};
cles.c
```

Pour appeler la fonction correspondant à la clé lue, il suffit d'exécuter la boucle suivante :

```
for (;;)
{
    int c = getchar();

    if (c == EOF)
        break;
    if (iscntrl(c))
        cle_fonct[c]();
    else
        err();
}
```

La pseudo-fonction `iscntrl`, qui teste si un caractère est un caractère de contrôle, est définie dans `ctype.h` (voir 7.2.1). La sortie de boucle est ici gérée de façon statique par le test

```
if (c == EOF)
```

Un problème se pose pour associer dynamiquement une fonction de terminaison à une clé, par exemple la fonction `fin` associée à la clé `C-f`. En effet, cela consiste à provoquer une sortie de la boucle principale depuis une fonction appelée dans cette boucle. La solution consiste à utiliser le mécanisme de sauvegarde et de restauration de contexte présenté en 7.9. On déclare un buffer de contexte `jmp_fin` visible à la fois dans la boucle et dans la fonction `fin`, qu'on initialise avant d'entrer dans la boucle de la façon suivante :

```
if (setjmp(jmp_fin) == 0)
    for (;;)
    {
        ...
    }
```

Il suffit alors de définir la fonction `fin` ainsi :

```
void fin(char c)
{
    longjmp(jmp_fin, 1);
}
```

6.5 Manipulation de listes

Une liste est composée d'éléments chaînés entre eux. Chaque élément, que nous appellerons ici une *cellule*, contient l'adresse de la cellule suivante dans la liste. Par exemple, la structure suivante permet de gérer des listes de chaînes de caractères.

```
struct cellule
{
    struct cellule *suivant;
    char *valeur;
};
typedef struct cellule *liste;
```

La figure 6.2 représente une liste formée à partir de ces cellules.

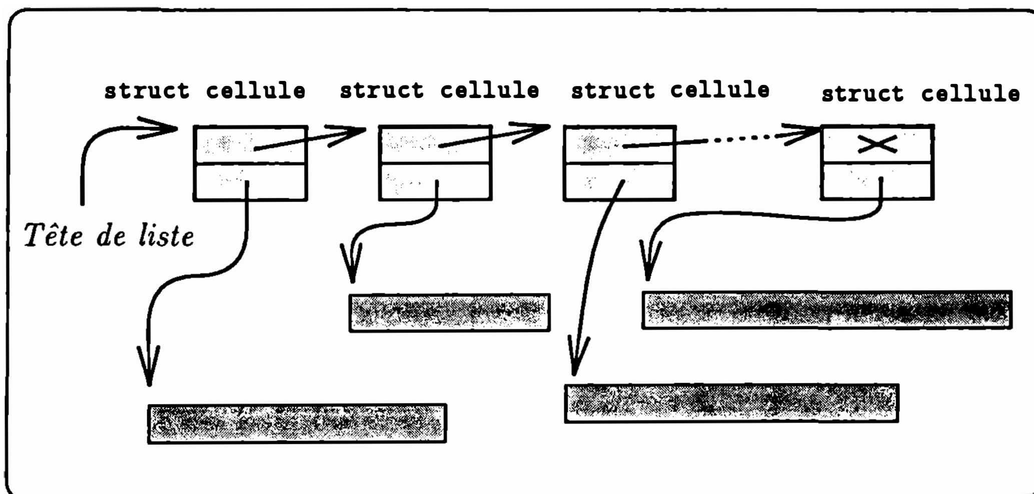


Figure 6.2: Chaînage de cellules

L'avantage de cette structure de données est d'être extensible durant l'exécution du programme. Chaque extension se traduit par la création dynamique d'une nouvelle cellule, puis par son chaînage sur la liste des cellules existantes. L'allocation dynamique de mémoire est réalisable au moyen de la fonction de bibliothèque `malloc`. Nous détaillons ce mécanisme dans le chapitre consacré à la bibliothèque standard, en 7.3. Pour allouer dynamiquement un objet, nous utilisons ici la pseudo-fonction `ALLOC`, définie page 221, à partir de la fonction `malloc`. Elle reçoit un type en paramètre et retourne l'adresse d'un nouvel objet de ce type. Par exemple

```
int *pi = ALLOC(int);
```

initialise le pointeur `pi` avec l'adresse d'un entier alloué dynamiquement à l'exécution.

La fonction `cellule_creer` effectue la création d'une nouvelle cellule. Elle reçoit en paramètre la chaîne de caractères à référencer. La zone mémoire occupée par cette chaîne doit également avoir été allouée dynamiquement.

La fonction `cellule_liberer` effectue le travail symétrique : la libération de l'espace occupé par une cellule. La libération est effectuée au moyen de la fonction de bibliothèque `free` (voir 7.3) libérant un objet alloué dynamiquement par un appel à `malloc`. On notera que pour libérer proprement une cellule, il est nécessaire de libérer auparavant la zone mémoire qu'elle référence.

La fonction `cellule_inserer` insère une nouvelle cellule dans la liste, à la suite d'une cellule passée en argument.

Enfin, la fonction `cellule_iterer` parcourt la liste, et pour chaque cellule, appelle une fonction $\langle f \rangle$ avec comme paramètre effectif la chaîne référencée par cette cellule. La fonction $\langle f \rangle$ est le second paramètre formel de la fonction `cellule_iterer`. Le passage de fonctions en paramètre est présenté en 6.4.3.

cellule.c

```
#include <stddef.h>

#include "exit_if.h"
#include "alloc.h"
#include "cellule.h"
#include "ERREURS.h"

cellule cellule_creer(char *chaine)
{
    cellule c = ALLOC(struct cellule);
    EXIT_IF(c==NULL, ERR_MALLOC);

    c->nom = chaine;
    c->suivant = NULL;
    return c;
}

void cellule_liberer(cellule c)
{
    if (c == NULL)
        return;
    if (c->nom != NULL)
        free(c->nom);
    free(c);
}

cellule cellule_inserer(cellule p, cellule c)
{
    c->suivant = p->suivant;
    p->suivant = c;
    return c;
}

void liste_iterer(cellule p, void (*f)(char *))
{
    while (p != NULL)
    {
        f(p->nom);
        p = p->suivant;
```

```
    }
}
═══════════════════════════════════ cellule.c ════════
```

Le fichier cellule.h contient la définition du type cellule, les prototypes des fonctions, et la définition d'une pseudo-fonction de récupération de la valeur d'une cellule.

```
════════ cellule.h ═══════════════════════════════════
#ifndef CELLULE_H
#define CELLULE_H

struct cellule
{
    struct cellule *suivant;
    char *valeur;
};
typedef struct cellule *cellule;

#define cellule_valeur(c) ((c)->valeur)

extern cellule cellule_creer(char *);
extern void cellule_liberer(cellule);
extern cellule cellule_inserer(cellule, cellule);
extern void liste_iterer(cellule, void (*)(char *));

#endif  /* CELLULE_H */
═══════════════════════════════════ cellule.h ════════
```

Le programme de test suivant construit une liste de chaînes en itérant la suite d'actions suivante :

- création d'une cellule initialisée avec une chaîne lue sur l'entrée standard;
- chaînage de la cellule créée à la fin de la liste.

Le premier élément est référencé par une cellule vide de nom premiere. Le pointeur derniere pointe sur le dernier élément de la liste. Lorsque la liste est vide, il pointe sur la cellule premiere. Cette technique, consistant à utiliser une cellule vide pour référencer le premier élément, permet d'éviter de traiter l'insertion dans une liste vide comme un cas particulier.

La lecture de la chaîne fin provoque la sortie de la boucle de saisie et l'affichage du contenu de la liste en itérant l'appel à la fonction puts de la bibliothèque standard; cette fonction recopie une chaîne de caractères sur la sortie standard. L'adresse de puts est convertie en pointeur de fonction void.

```
════════ tliste.c ════════════════════════════════════
#include <stddef.h>
#include <string.h>
#include <stdio.h>

#include "cellule.h"
#include "lire_chaine.h"

#define FIN    "fin"
#define PROMPT "->"
```

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

main()
{
    cellule premiere = cellule_creer(NULL);
    cellule derniere = premiere;

    for (;;)
    {
        cellule c = cellule_creer(lire_chaine(PROMPT));

        if (strcmp (cellule_valeur(c), FIN) == 0)
        {
            cellule_liberer(c);
            break;
        }
        derniere = cellule_inserer(derniere, c);
    }
    liste_iterer(premiere->suivant, (void (*)(char *)) puts);
}
```
tliste.c

La lecture d'une chaîne de caractères est effectuée au moyen d'une fonction appelée `lire_chaine` qui réalise :

- l'affichage sur la sortie standard d'un prompt passé en paramètre;
- la lecture sur l'entrée standard d'une chaîne de caractères;
- l'allocation dynamique d'une zone mémoire et la recopie de la chaîne lue dans cette zone.

La fonction retourne l'adresse de la chaîne allouée dynamiquement. On trouvera page 222 une définition de la fonction `lire_chaine` permettant la lecture de chaînes de caractères de longueur quelconque.

```
$ tliste
-> Ceci est la premiere \
chaine lue
-> Voici la seconde
-> Une derniere pour terminer
-> fin
Ceci est la premiere chaine lue
Voici la seconde
Une derniere pour terminer
$
```

Remarque 24 *Nous nous sommes volontairement limité à un exemple très simple de listes simplement chaînées. Dans la pratique, on a en général besoin de listes doublement chaînées avec gestion d'une position courante et insertion à n'importe quel endroit de la liste.*

Chapitre 7

La bibliothèque standard

Le langage C *pur* se limite aux constructions présentées dans les premiers chapitres : déclarations, expressions, instructions et blocs, structures de contrôles, et fonctions. Il n'y a, en particulier, pas d'instruction d'entrées-sorties. Celles-ci sont implémentées sous la forme de fonctions contenues dans la bibliothèque standard du langage.

Initialement, cette bibliothèque était totalement dépendante de l'implémentation et pouvait varier de façon non négligeable d'un système d'exploitation à l'autre (UNIX, MSDOS, VMS...) et même d'une version à une autre d'un même système, de BSD à SYSTEM V en passant par toutes les implémentations hybrides. Cela a constitué pendant longtemps la difficulté majeure de portage des programmes C. La norme ANSI constitue sur ce point un progrès considérable. Elle comporte une liste précise de toutes les fonctions devant être définies dans l'environnement standard, avec pour chacune, son prototype et sa sémantique. La plupart de ces fonctions étaient déjà, dans les environnements UNIX, des standard de faits.

7.1 Les entrées-sorties standard

7.1.1 Principes généraux

Une opération d'écriture est un transfert de données effectué depuis une zone mémoire du programme vers une unité physique : terminal, unité de disque, imprimante, etc. L'opération de lecture est l'opération symétrique, c'est-à-dire un transfert de données depuis une unité physique vers la mémoire.

Ces opérations de très bas niveau sont réalisées par le système d'exploitation lorsqu'un programme exécute une requête d'entrée-sortie, encore appelée entrée-sortie *de bas niveau*. Sous UNIX, ces requêtes correspondent aux appels système **WRITE(2)** et **READ(2)**, sur lesquels nous reviendrons plus loin. Les fonctions d'entrées-sorties de la bibliothèque standard, encore appelées *entrées-sorties de haut niveau*, constituent une interface avec les entrées-sorties de bas niveau. Elles intègrent deux mécanismes distincts :

- le *formatage* des données;

– la mémorisation des données dans une mémoire intermédiaire ou *mémoire tampon* (*buffer* dans la terminologie anglo-saxonne).

On parlera d'entrées-sorties **mémorisées** lorsque les données sont stockées dans une mémoire tampon, et d'entrées-sorties **immédiates** sinon. L'argument principal d'une opération d'entrée-sortie de haut niveau (que nous appellerons simplement entrée-sortie dans la suite de ce chapitre) est un **flot** (de l'anglo-saxon *stream*). Un flot fait référence à l'ensemble des informations relatives à une entrée-sortie en cours : nature de l'entrée-sortie, fichier sur lequel elle porte, mémoire tampon, position courante dans le fichier... Cet ensemble d'informations, codé par une structure, est identifié par l'identificateur de type `FILE`. Dans un programme, un flot est déclaré de type `FILE *`.

Trois flots sont prédéfinis au démarrage d'un programme :

– `stdin` : initialisé en lecture sur l'entrée standard (en général le clavier du terminal);
– `stdout` : initialisé en écriture mémorisée sur la sortie standard (en général l'écran du terminal);
– `stderr` : initialisé en écriture immédiate sur la sortie erreur standard (en général l'écran du terminal).

Les flots `stdin` et `stdout` sont utilisables implicitement au moyen des fonctions `scanf` et `printf`. On peut initialiser d'autres flots en *ouvrant* une entrée-sortie au moyen de la fonction `fopen`. Diverses fonctions, comme par exemple `fprintf`[1], `fscanf`, `fflush`... permettent de lire et d'écrire sur ces flots.

Les définitions nécessaires à la mise en œuvre de ces fonctions (définition du type `FILE`, déclaration des flots `stdin`, `stdout` et `stderr`...) se trouvent dans le fichier en-tête `stdio.h`. Il est par conséquent nécessaire de placer la directive

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

en tête de tout fichier dans lequel sont effectuées des entrées-sorties standard. Dans la suite de ce chapitre, pour chaque fonctionnalité présentée, les noms des fichiers en-tête nécessaires sont rappelés en début de section.

7.1.2 Mécanismes de formatage

`<stdio.h>`

Un formatage est la conversion de données internes du programme en chaînes de caractères, en général dans le but de les imprimer. Par exemple, pour être affichée, la valeur décimale 65 doit être convertie en la suite de caractères `6 5`. Cette même valeur peut être interprétée comme le code du caractère `A`.

Le formatage s'effectue à partir d'un **format** et de la liste des valeurs à formater. Le résultat est une chaîne de caractères. Un format est également une chaîne de caractères contenant

– des caractères ordinaires recopiés tels quels dans la chaîne résultat;
– des spécifications de conversion, suites de caractères commençant par le symbole `%`.

[1] Le `f` préfixant le nom de la fonction signifie *file*, celui le suffixant signifie *formatted*.

Chaque spécification de conversion est associée à un élément de la liste des valeurs à formater. Par exemple, `%c` signifie que la valeur associée doit être interprétée comme le codage d'un caractère; `%d` (resp. `%x`) signifie qu'elle doit être interprétée comme un entier et écrite en base 10 (resp. en base 16). Le programme `format1.c` montre plusieurs manières d'afficher une valeur.

```
========== format1.c ==========================================
#include <stdio.h>
#include <stdlib.h>

main(int argc, char *argv[])
{
    int nombre = (argc == 2) ? atoi(argv[1]) : 0;

    printf("Si nombre est interprete comme\n");
    printf(" -un caractere : il s'ecrit '%c';\n",
           nombre);
    printf(" -un entier: il s'ecrit '%d' en base 10,\n",
           nombre);
    printf("                         et '%x' en base 16.\n",
           nombre);
}
=========================================== format1.c ==========
```

Lors de son exécution (voir figure 7.1), on obtient successivement la représentation du caractère, et son code ASCII en décimal et en hexadécimal. La seconde exécution est effectuée avec la valeur 167 qui ne correspond pas à un code ascii standard, mais à un caractère étendu de la norme ISO-8859.

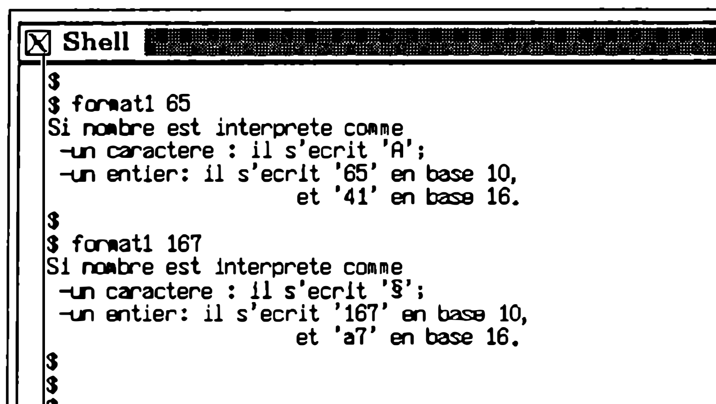


Figure 7.1: Exécution du programme `format1`

Une spécification de conversion peut contenir les informations suivantes :

1) le type de la donnée : caractère, entier (sans signe ou signé, court ou long), réel ou chaîne de caractères;
2) la notation du résultat : base 8, 10 ou 16 pour un entier, nombre de chiffres après la virgule, notation flottante ou exponentielle pour un réel;
3) le cadrage du résultat : taille du champ, cadrage dans le champ...

Le tableau 7.1 fournit les différentes spécifications qu'il est possible de construire. Chaque ligne décrit les spécifications correspondant à un type de pa-

Type du paramètre	Notation	Options de cadrage et précision								
		(1) -	(2) +	(3) ␣	(4) 0	(5) *n*	(6) .*n*	(7) l	(8) #	
Caractère	—	×				×				c
Entier sans signe	Décimale	×			×	×		×		u
	Octale	×			×	×		×	×	o
	Hexadéc.	×			×	×		×	×	x, X
Ent. signé	Décimale	×	×	×	×	×		×		d, i
Réel	Flottante	×	×	×	×	×	×		×	f
	Exponen.	×	×	×		×	×		×	e, E
	Générale	×	×	×	V	×	×		×	g, G
Chaîne	—	×				×				s
Pointeur	*non spécif.*									p

× : option prise en compte
V : option prise en compte avec option # seulement

(1) : cadré à gauche
(2) : préfixé avec le signe + si le nombre est positif – exclut (3)
(3) : préfixé avec un espace si le nombre est positif – exclut (2)
(4) : complété à gauche par des zéros
(5) : nombre minimum de caractères ou * (longueur en paramètre)
(6) : précision décimale ou * (longueur en paramètre)
(7) : le paramètre est un entier long
(8) : %o, nombre préfixé par 0
%x (%X), nombre préfixé par 0x (0X)
%f, %e ou %E, et %g, point décimal toujours présent

%x : résultat écrit avec les caractères 0 ... 9 a b c d e f
%X : résultat écrit avec les caractéres 0 ... 9 A B C D E F
%e : résultat de la forme [-]*n.dddddd*e±*xx*
%E : résultat de la forme [-]*n.dddddd*E±*xx*
%g (%G) : équivalent à %e (%E) si l'exposant est inférieur à -4 ou supérieur ou égale à la précision, et à %f sinon

Tableau 7.1: Spécifications de conversion de format

ramètre particulier. Une ligne se décompose en plusieurs sous-lignes lorsqu'il existe plusieurs notations différentes pour un même type.

Une spécification commence par un %, se continue par d'éventuelles options de cadrage et de précision, choisies parmi les options prises en compte pour ce type, et se termine par le caractère de la dernière colonne, qui code à la fois le type du paramètre et la notation du résultat. Les options s'utilisent dans l'ordre où elles apparaissent dans le tableau (colonnes (1) à (8)). Les colonnes options (2) et (3) ne peuvent être utilisées simultanément.

Par exemple, la spécification `%-␣05ld` signifie :

> *"entier signé long, à afficher en notation décimale, cadré à gauche, le premier caractère étant une espace si le nombre est positif, et écrit sur cinq caractères au minimum"*

Le programme suivant montre quelques exemples de formatage :

```
═══════ format2.c ═══════
#include <stdio.h>

main()
{
    short  entier_court = 1968;
    long   entier_long = 56849915;
    double reel = 202.199155556;
    char   *chaine = "seize caracteres";

    printf("\n Type :\n");
    printf("   entier_court %d, comme un reel %f\n",
            entier_court, entier_court);
    printf("   chaine \"%s\", comme un entier %d\n",
            chaine, chaine);

    printf("\n Notation :\n");
    printf("   entier_long %ld en octal %l#o et hexa %l#x\n",
            entier_long, entier_long, entier_long);
    printf("   reel %.10f, arrondi a %.4f, ou encore %.6e\n",
            reel, reel, reel);

    printf("\n Cadrage :\n");
    printf("   entier_court %d\n", entier_court);
    printf("     -sur 6 caracteres (%8d),\n", entier_court);
    printf("     -cadre a gauche (%-8d),\n", entier_court);
    printf("     -complete par des zeros (%08d),\n",
           entier_court);
    printf("   reel %.10f\n", reel);
    printf("     -arrondi, a gauche, avec signe (%-+8.2f)\n",
           reel);
}
═══════ format2.c ═══════
```

Voici le résultat de l'exécution de ce programme :

```
$ format2

 Type :
   entier_court 1968, comme un reel 0.000000
```

```
    chaine "seize caracteres", comme un entier 8848

 Notation :
    entier_long 56849915 en octal 0330672773 et hexa 0x36375fb
    reel 202.1991555560, arrondi a 202.1992, ou encore 2.021992e+02

 Cadrage :
    entier_court 1968
      -sur 6 caracteres (    1968),
      -cadre a gauche (1968    ),
      -complete par des zeros (00001968),
    reel 202.1991555560
      -arrondi, a gauche, avec signe (+202.20 )
```

La spécification de taille peut être remplacée par le caractère `*`. Dans ce cas, le paramètre suivant est utilisé comme spécificateur de taille. Par exemple, la spécification `%*d` utilise deux paramètres :

- le premier pour fixer le nombre minimal de caractères de cette spécification,
- le second comme valeur à formater en notation décimale.

On peut tester ce mécanisme avec le programme suivant :

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

#include "pluriel.h"

main(int argc, char *argv[])
{
    int l1 = argc > 1 ? atoi(argv[1]) : 1;
    int l2 = argc > 2 ? atoi(argv[2]) : 1;

    printf("Formatage de %d sur %d caractere%s minimum: ",
           l1, l2, PLURIEL(l2));
    printf("%0*d\n", l2, l1);
}
format_etoile.c
```

Le nombre affiché est complété à droite avec des zéros.

```
$ format_etoile 230 5
Formatage de 230 sur 5 caracteres minimum: 00230
$ format_etoile 230 16
Formatage de 230 sur 16 caracteres minimum: 0000000000000230
$ format_etoile 230 1
Formatage de 230 sur 1 caractere minimum: 230
$
```

Le formatage peut être effectué au moyen des trois fonctions :

- `sprintf` : le résultat est copié à une adresse mémoire;
- `printf` : le résultat est envoyé sur la sortie standard;
- `fprintf` : le résultat est envoyé dans un fichier associé à un flot.

La syntaxe de `sprintf` est :

int sprintf(char *⟨*résultat*⟩, const char *⟨*format*⟩, ...)

Le paramètre ⟨*résultat*⟩ est l'adresse d'un vecteur de caractères dans lequel la fonction `sprintf` place le résultat du formatage. Le second paramètre est le format. Vient ensuite la liste éventuelle des valeurs à formater. La valeur de retour est la longueur de la chaîne résultant du formatage. Nous avons déjà rencontré de nombreux exemples d'utilisation de `printf`, et la fonction `fprintf` est détaillée en 7.1.4. Voici un exemple d'utilisation de `sprintf` :

```
sprintf(format_erreur, "%s: %%s\\n", argv[0])
```

L'exécution de cette expression construit la chaîne

⟨*cmd*⟩:%s\n

à l'adresse `format_erreur`. Le principal problème consiste à fixer la taille à priori de la zone mémoire recevant le résultat du formatage.

Une des difficultés majeures du formatage réside dans la compréhension des mécanismes de *quotation*, c'est-à-dire l'opération consistant à bloquer l'évaluation d'un caractère au moyen de caractères spéciaux, comme ', ", ou \. Un exercice classique de manipulation de quotation consiste à écrire un programme *auto reproducteur*, c'est-à-dire dont le source et le résultat d'exécution sont rigoureusement identiques. En voici un exemple qu'on pourra étudier.

```
auto.c
char *f = "char *f = %c%s%c;
char c = '%c';

main()
{
    printf(f, c, f, c, c);
}
";
char c = '"';

main()
{
    printf(f, c, f, c, c);
}
auto.c
```

Ce programme, en apparence très simple, contient une définition de format décrivant son propre source. Le résultat de son exécution est la copie exacte de son source :

```
$ auto
char *f = "char *f = %c%s%c;
char c = '%c';

main()
{
```

```
    printf(f, c, f, c, c);
}
";
char c = '"';

main()
{
    printf(f, c, f, c, c);
}
```

La comparaison du résultat de son exécution avec son source peut être effectuée au moyen de la commande **DIFF (1)** : les deux textes sont rigoureusement identiques.

```
$ auto > AUTO
$ diff auto.c AUTO
$
```

7.1.3 Ouverture et fermeture de flot

<stdio.h>

L'initialisation d'un nouveau flot est effectuée par la fonction `fopen` :

`FILE *fopen(const char *⟨`*chemin*`⟩, const char *⟨`*mode-d'accès*`⟩)`

Le paramètre *chemin* est le nom du fichier à associer au nouveau flot. Le second paramètre décrit le type d'accès qu'on veut effectuer : lecture et/ou écriture, et positionnement. C'est une chaîne formée de l'un des trois caractères `r`, `w` et `a`, suivi éventuellement du caractère `+`. Lors d'une ouverture en écriture, le fichier peut, soit être créé s'il n'existe pas, soit encore être initialisé, c'est-à-dire ramené à une taille nulle, s'il existe déjà. Les modes `r+`, `w+` et `a+` correspondent à trois façons différentes d'initialiser le flot en mode lecture et écriture. Les différents modes d'ouverture sont résumés dans le tableau 7.2. D'autre part, le suffixe `b` concaténé au mode signifie *entrée-sortie en mode binaire*, le mode par défaut étant le mode *texte*[2].

Lorsque l'ouverture s'effectue sans erreur, la fonction retourne la référence à un flot. Il y a diverses situations d'erreur : chemin incorrect, fichier protégé... Dans ces cas-là, la fonction `fopen` retourne la valeur `NULL`. Par exemple, l'instruction

`FILE *f_init = fopen(init_file_name, "r");`

déclare un flot de nom `f_init` et l'ouvre en lecture sur le fichier dont le nom est contenu dans la chaîne `init_file_name`. Si ce fichier n'existe pas, ou n'est pas accessible, la valeur retournée par `fopen` est la valeur `NULL`.

L'opération symétrique de l'ouverture d'un flot est sa fermeture, réalisée par la fonction `fclose`. Cette fonction vide la mémoire tampon du flot et ferme

[2]Cette distinction n'a pas de sens sous UNIX où tous les fichiers sont gérés de façon homogène.

Mode d'accès	Paramètre	Position	Comportement si le fichier existe	Comportement si le fichier n'existe pas
Lecture	`r`	début	—	erreur
Ecriture	`w`	début	initialisation	création
	`a`	fin	—	création
Lecture et écriture	`r+`	début	—	erreur
	`w+`	début	initialisation	création
	`a+`	fin	—	création
	Suffixe `b`	: entrée-sortie binaire		

Tableau 7.2: Modes d'ouverture d'un flot d'entrées-sorties

la communication. Le descripteur de flot pourra être alloué à nouveau par la fonction `fopen`.

Il est possible de redéfinir un flot déjà initialisé, en changeant le fichier qui lui est associé. Cette opération correspond à la redirection d'entrée-sortie très classique sous UNIX. Pour cela, on utilise la fonction `freopen`, dont la syntaxe est la suivante :

```
FILE *freopen(const char *⟨chemin⟩,
              const char *⟨mode⟩, FILE *⟨flot⟩)
```

Si le descripteur de flot ⟨*flot*⟩ est actif, la fonction `freopen` commence par refermer le flot associé. Elle effectue ensuite l'ouverture d'un nouveau flot, associé au fichier ⟨*chemin*⟩, et selon le mode décrit par le paramètre ⟨*mode*⟩.

Si l'ouverture réussit, la référence au nouveau flot est placée dans le descripteur ⟨*flot*⟩, et la fonction `reopen` retourne le descripteur ⟨*flot*⟩. Sinon, elle retourne la valeur `NULL`. L'exemple suivant montre une redirection du flot sortie standard dans un fichier de nom `execution.log`.

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

#define NOM_LOG  "execution.log"

main()
{
    freopen(NOM_LOG, "w", stdout);
    printf("   ) Debut d'execution de reopen  )\n");
    printf("  /               ...              /\n");
    printf(" ( Fin d'execution de reopen ___(\n");
}
reopen.c
```

La première commande exécutée dans la session suivante permet de vérifier qu'aucun fichier suffixé par `.log` n'existe. C'est donc l'exécution de la commande `reopen` qui le crée. On vérifie au moyen de la commande **CAT(1)** qu'il contient toutes les lignes écrites sur la sortie standard.

```
$ ls *.log
*.log not found
$
$ reopen; ls *.log
execution.log
$
$ cat execution.log
   ) Debut d'execution de reopen  )
  /            ...               /
 ( Fin d'execution de reopen ___(
$
```

Un programme peut délibérément ignorer les redirections d'entrées-sorties standard effectuées depuis l'interprète de commandes, en effectuant une réouverture du terminal auquel il est attaché. Par exemple sous UNIX, le terminal attaché étant représenté par le fichier spécial `/dev/tty`, l'exécution de

```
freopen("/dev/tty", "w", stdout)
```

rend un programme insensible aux redirections de sa sortie standard.

La commande `noredir` suivante recopie une chaîne reçue en argument (ou son entrée standard dans le cas où il n'y a pas d'argument) sur sa sortie standard. Auparavant, elle effectue une réouverture de cette sortie standard sur le terminal attaché au processus.

```
============ noredir.c ============
#include <stdio.h>
#define TTY "/dev/tty"

main(int argc, char **argv)
{
    freopen(TTY, "w", stdout);

    if (argc > 1)
    {
        while(++argv, --argc)
            printf("%s ", *argv);
        printf("\n");
    }
    else
    {
        for (;;)
        {
            int c = getchar();

            if (c == EOF)
                break;
            putchar(c);
        }
    }
}
============ noredir.c ============
```

L'exemple d'exécution suivant illustre l'impossibilité de détourner les sorties du programme `noredir`, en effectuant une redirection sur le fichier spécial `/dev/null` (sous UNIX, le fichier spécial `/dev/null` est associé à un périphérique qui fait disparaître toutes les données qu'il reçoit – voir 10.1.1). Pour plus de clarté les messages affichés par le programme ou par le système sont en caractères penchés.

```
$ echo Les ecrits s\'envolent parfois...
Les ecrits s'envolent parfois...
$ echo Les ecrits s\'envolent parfois... > /dev/null
$
$ noredir > /dev/null
Des mots
Des mots
qu'on ne peut detourner...
qu'on ne peut detourner...
$ noredir Des mots qu\'on ne peut detourner... > /dev/null
Des mots qu'on ne peut detourner...
$
```

La fermeture d'un flot s'effectue au moyen de la fonction `fclose`:

`int fclose(FILE *⟨flot⟩)`

Elle retourne la constante `EOF` si une erreur s'est produite, et zéro sinon.

7.1.4 Fonctions de lecture et d'écriture

`<stdio.h>`

L'écriture sur un flot peut être effectuée au moyen de la fonction `fprintf`, dont la syntaxe est la suivante :

`int fprintf(FILE *⟨flot⟩, const char *⟨format⟩, ...)`

Elle enchaîne la suite d'actions suivante :

– formatage de la liste éventuelle de paramètres conformément à la spécification ⟨*format*⟩;
– si l'écriture est mémorisée
  * mémorisation du résultat dans la mémoire tampon du flot,
  * éventuellement, vidage du tampon et écriture du résultat sur l'unité physique associée;

  sinon
  * écriture du résultat sur l'unité physique associée.

Par exemple

```
fprintf(stderr, "%s: invalid file name\n", argv[1])
```

construit une chaîne de caractères de la forme

nom: `invalid file name`

et l'écrit sur la sortie erreur standard. La fonction `printf` est, en fait, une forme abrégée de

```
fprintf(stdout,...)
```

La lecture sur un flot d'entrée s'effectue au moyen de la fonction `fscanf` dont la syntaxe est la même que la fonction `fprintf`. La fonction `scanf` est, en fait, une forme abrégée de

```
fscanf(stdin,...)
```

Les paramètres de la fonction `fscanf` doivent être passés par référence; pour lire un entier `i` sur l'entrée standard, on écrit

```
scanf("%d", &i)
```

Si `chn` est un vecteur de caractères, la lecture d'une chaîne peut s'écrire

```
scanf("%s", chn)
```

En principe, il est possible de spécifier des formats d'entrée plus complexes que ceux suggérés sur ces exemples. Mais, l'utilisation de `fscanf` devient vite délicate, et il n'est pas rare que des formats plus sophistiqués provoquent une terminaison sur exception.

Il existe un certain nombre d'autres fonctions de lecture ou d'écriture. Une des plus utilisées est la fonction `getchar` qui retourne un caractère acquis sur l'entrée standard. La fonction `getc` effectue le même travail sur un flot d'entrée passé en paramètre (l'expression `getchar()` est équivalente à `getc(stdin)`).

Les fonctions symétriques sont `putchar` et `putc`. Il existe également des fonctions manipulant les chaînes de caractères (`gets`, `fgets`, `puts` et `fputs`), et des fonctions de lecture et écriture sans formatage (`fread` et `fwrite`).

Nous allons traiter un exemple plus complet utilisant les fonctions `fopen`, `fgets`, `fprintf` et `fclose`. Le premier programme construit un fichier dont le nom est reçu en argument. Il le remplit avec des instructions C.

```
sq_genere.c
#include <stdio.h>
#include <stddef.h>

main(int argc, char *argv[])
{
    FILE *flot;

    if (argc != 2)
        exit(1);
    /* Creation */
    if ((flot = fopen(argv[1], "w")) == NULL)
    {
        fprintf(stderr, "Impossible de creer %s\n", argv[1]);
        exit(1);
    }
    printf("Construction du fichier %s\n", argv[1]);
    fprintf(flot, "main()\n");
    fprintf(flot, "{\n");
    fprintf(flot, "    printf(\"fichier %s\\n\");\n",argv[1]);
    fprintf(flot, "}\n");
    fclose(flot);
}
sq_genere.c
```

S'il n'est pas possible de créer le fichier dont le chemin est passé en paramètre, la fonction `fopen` retourne la valeur `NULL`. L'exemple suivant illustre ce cas, le paramètre `"."` étant le nom du répertoire courant sous UNIX, inaccessible en écriture.

```
$ sq_genere essai.c
Construction du fichier essai.c
$
$ cat essai.c
main()
{
    printf("fichier essai.c\n");
}
$
$ sq_genere .
Impossible de creer .
$
```

Le programme `sq_lit` permet de relire le fichier qu'on vient de créer. Il utilise la fonction `fgets` qui s'emploie avec trois paramètres :

1) l'adresse du vecteur où copier la chaîne lue;
2) le nombre maximum de caractères à lire (la lecture s'interrompt normalement sur le premier caractère `\n` rencontré);
3) le flot sur lequel est effectuée la lecture.

```
========== sq_lit.c ==========
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>

main(int argc, char *argv[])
{
    char buffer[256];
    FILE *flot;

    if (argc != 2)
        exit(1);

    if ((flot = fopen(argv[1], "r")) == NULL)
    {
        fprintf(stderr, "Impossible de lire %s\n", argv[1]);
        exit(1);
    }
    printf("Lecture de %s\n", argv[1]);
    while (fgets(buffer, sizeof buffer, flot) != NULL)
        fputs(buffer, stdout);
    fclose(flot);
}
========== sq_lit.c ==========
```

Voici plusieurs essais d'utilisation de `sq_lit`, suivis de la compilation et de l'exécution du programme généré.

```
$ sq_lit xxxx
```

```
Impossible de lire xxxx
$
$ sq_lit essai.c
Lecture de essai.c
main()
{
    printf("fichier essai.c\n");
}
$
$ gcc -g essai.c -o essai
$ essai
fichier essai.c
$
```

7.1.5 Accès séquentiel et accès direct

`<stdio.h>`

Par défaut, les fonctions que nous avons citées dans la section précédente travaillent en mode séquentiel. Chaque lecture ou écriture s'effectue à partir d'une position courante, et incrémente cette position du nombre de caractères lus ou écrits. La position courante est initialisée en fonction du mode d'ouverture, comme indiqué dans le tableau 7.2. Il est possible de connaître et de modifier la position courante au moyen des fonctions :

int fseek(FILE * ⟨*flot*⟩, long ⟨*déplacement*⟩, int ⟨*origine*⟩)
long ftell(FILE * ⟨*flot*⟩)

La fonction `ftell` retourne la position courante exprimée en octets. La fonction `fseek` permet de modifier la position courante sur le flot ⟨*flot*⟩. Le second paramètre indique un déplacement exprimé en octets et le troisième l'origine du déplacement. Les trois valeurs possibles de ⟨*origine*⟩ sont :

- SEEK_SET : à partir du début du fichier;
- SEEK_CUR : à partir de la position courante;
- SEEK_END : à partir de la fin du fichier;

Dans une implémentation non standard, on utilisera les constantes 0, 1 et 2.

Voici quelques exemples d'utilisation de cette fonction. L'expression

`fseek(donnees, 16L, SEEK_CUR)`

incrémente la position courante du flot `donnees` de 16 caractères, et l'expression

`fseek(donnees, -256L, SEEK_END)`

la ramène à 256 octets de la fin du fichier. L'expression

`position = ftell(result)`

sauvegarde la position courante du flot `result`, et l'expression

`fseek(result, position, SEEK_SET)`

permet de la restaurer.

Il existe également un couple de fonctions utilisables même lorsque la position dans le fichier est une quantité supérieure à la taille maximale d'un entier long. Le prototype de ces fonctions est

```
int fgetpos(FILE *⟨flot⟩, fpos_t *⟨position⟩)
int fsetpos(FILE *⟨flot⟩, fpos_t *⟨position⟩)
```

Le paramètre ⟨*position*⟩ est utilisé, dans le premier cas pour recevoir la position courante dans ⟨*flot*⟩, et dans le second pour la modifier.

7.1.6 Contrôle de la mémorisation

`<stdio.h>`

La mémoire tampon associée à un flot de sortie est vidée lorsqu'elle est pleine; elle peut aussi être vidée au moyen de la fonction `fflush` :

```
int fflush(FILE *⟨flot⟩)
```

Le tampon est également automatiquement vidé lors de la fermeture du flot par `fclose`.

Lorsqu'un programme termine, les entrées-sorties actives sont automatiquement fermées, et par conséquent, les tampons vidés. Cela n'est pas le cas si l'exécution est interrompue de façon anormale (interruption clavier, erreur mémoire...). Le programme suivant illustre ce comportement.

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

#define TAMPON "TAMPON"

main()
{
    FILE *f = fopen(TAMPON, "w");

    fprintf(f, "Debut du fichier " TAMPON "\n");
    fflush(f);

    fprintf(f, "Fin du fichier " TAMPON "\n");

    printf("Exception ? (o/n) : ");
    if (getchar() == 'o')
        *((char *) 0) = 0;
}
tampon.c
```

Lors de la seconde exécution, le programme tente d'écrire à l'adresse zéro après le second appel à la fonction `printf`, ce qui provoque une terminaison anormale. Le fichier ne contient dans ce cas que la première ligne, celle dont l'écriture physique a été forcée par la fonction `fflush`.

```
$ tampon
Exception ? (o/n) : n
$
$ cat TAMPON
```

```
Debut du fichier TAMPON
Fin du fichier TAMPON
$
$ tampon
Exception ? (o/n) : o
Segmentation fault (core dumped)
$
$ cat TAMPON
Debut du fichier TAMPON
$ rm core
$
```

On peut constater sur cet exemple que les entrées-sorties standard obéissent à un mécanisme de synchronisation particulier. En effet la chaîne

`"Exception ? (o/n) : "`

a été affichée, bien qu'on n'ait pas forcé explicitement le vidage du tampon de `stdout`. Le flot de sortie standard est automatiquement vidé :

- sur écriture d'un `\n`,
- lors de l'exécution d'une fonction de lecture sur `stdin`.

C'est donc l'appel à la fonction `getchar` qui a provoqué le vidage du tampon de `stdout`. Le remplissage de la mémoire tampon d'un flot d'entrée est provoqué, lorsqu'elle est vide, par l'exécution d'une fonction de lecture. Le nombre d'octets réellement transférés dépend de l'unité physique associée :

- sur un terminal, c'est le nombre de caractères tapés jusqu'au RETURN;
- sur un fichier, le tampon est rempli tant qu'il reste des octets à lire.

Prenons l'exemple d'une boucle répétant des lectures au moyen de la fonction `getchar`, la lecture se faisant par défaut au terminal. Si on entre la suite de caractères `abcde` validée par RETURN, la mémoire tampon reçoit six caractères, le sixième étant `\n`. Les six premiers appels à `getchar` lisent chacun un caractère dans la mémoire tampon, et le septième appel génère une lecture physique. Si ce même programme est exécuté avec son entrée standard redirigée dans un fichier, tant qu'il reste suffisamment de caractères dans le fichier, la mémoire tampon est remplie à chaque lecture physique.

L'utilisation simultanée de plusieurs fonctions de lecture peut entrainer un comportement du programme en apparence anormal. En effet, certaines fonctions lisent tous les caractères présents dans le tampon, comme `getchar` ou `fread`, alors que d'autres peuvent défiler certains caractères considérés comme des séparateurs, comme `scanf` ou `gets`.

Pour éviter ce genre de problèmes, une solution simple consiste à utiliser une boucle sur `getchar` ou `fgetc` pour défiler successivement tous les caractères d'un fichier, et `gets` ou `fgets` pour lire des lignes terminées par le caractère `\n`.

Si on désire uniformiser les lectures, par exemple en n'utilisant que la fonction `getchar`, il faut prendre la précaution de défiler les éventuels séparateurs, comme `\n`, lorsque ceux-ci sont inutiles. Par exemple, si on effectue une boucle lisant un caractère et appelant une fonction associée à ce caractère, on pourra écrire :

```
for (;;)
{
    int c = getchar();

    switch (c)
    {
      /* Defilement des separateurs */
      case '\n':
      case ' ':
      case '\t':
        break;

      /* Caracteres inconnus */
      default:
        erreur();
        break;

      case ...
```

Il existe d'autres manières d'effectuer des lectures. Citons par exemple

- les entrées-sorties de bas niveau, présentées dans le chapitre 10;
- l'utilisation de générateurs d'automates, comme les outils UNIX standard **LEX**(1) et **YACC**(1), ou leurs équivalents dans le domaine public comme `flex` ou `bison`;
- des bibliothèques spécialisées, comme la bibliothèque `readline`, bibliothèque du domaine public provenant du projet *GNU* et intégrant des mécanismes très sophistiqués comme l'édition vidéo de l'historique des précédentes commandes lues ou les substitutions de méta-caractères.

7.2 Manipulation de caractères et de chaînes

7.2.1 Type d'un caractère

`<ctype.h>`

On dispose d'un ensemble de pseudo-fonctions, définies dans le fichier en-tête `ctype.h`, testant le type d'un caractère : lettre, chiffre, espace... Bien sûr ces tests sont très simples à mettre en œuvre au moyen d'expressions logiques. Il est cependant préférable d'utiliser les fonctions proposées dans `ctype.h` pour deux raisons : leur portabilité et leur efficacité. Elle sont généralement implémentées au moyen d'un masquage effectué sur un vecteur de caractères (voir le fichier `<ctype.h>`), et offrent la garantie d'un résultat identique quel que soit l'environnement : prise en compte des définitions de caractères locaux (ß en Allemagne, ç en France...), implémentations n'utilisant pas le codage ASCII des caractères (EBCDIC par exemple), etc. De plus, l'utilisation de ces fonctions accroît la lisibilité du programme. On trouvera la liste et la description des fonctions de test dans le tableau 7.3.

Deux fonctions de conversion sont également définies :

```
toupper  :  minuscule ⟶ majuscule
tolower  :  majuscule ⟶ minuscule
```

<table>
<tr><th>Nom</th><th>Ensemble testé</th><th>Définition de base</th></tr>
<tr><td><code>isalpha</code></td><td>lettres</td><td><code>A-Z a-z</code></td></tr>
<tr><td><code>isupper</code></td><td>majuscules</td><td><code>A-Z</code></td></tr>
<tr><td><code>islower</code></td><td>minuscules</td><td><code>a-z</code></td></tr>
<tr><td><code>isdigit</code></td><td>chiffres</td><td><code>0-9</code></td></tr>
<tr><td><code>isxdigit</code></td><td>chiffres hexadécimaux</td><td><code>0-9 A-F a-f</code></td></tr>
<tr><td><code>isalnum</code></td><td>lettres ou chiffres</td><td><code>A-Z a-z 0-9</code></td></tr>
<tr><td><code>isspace</code></td><td>espaces</td><td><code>\v \n \t \f \r ␣</code></td></tr>
<tr><td><code>ispunct</code></td><td>ponctuation: caractères affichables autres que les précédents</td><td><code>!"#$%&'()*+,-./:
;<=>?@[\]^_`{|}~</code></td></tr>
<tr><td><code>isprint</code></td><td colspan="2">lettres, chiffres, ponctuation ou espaces</td></tr>
<tr><td><code>isgraph</code></td><td colspan="2">lettres, chiffres, ou ponctuation</td></tr>
<tr><td><code>iscntrl</code></td><td>caractères de contrôle</td><td><code>\000-\037 \177</code></td></tr>
</table>

Tableau 7.3: Type d'un caractère : `ctype.h`

Remarque 25 *Les fonction* `isascii` *testant si un caractère est un code* ASCII *et* `toascii` *convertissant un code en un code* ASCII, *que l'on retrouve dans la plupart des implémentations* ASCII, *n'ont pas été intégrées dans la norme* ANSI *pour des raisons de compatibilités avec les autres implémentations.*

L'exemple suivant est un filtre qui convertit la première lettre de chaque phrase en caractère majuscule et les autres lettres en caractères minuscules.

```
majuscule.c
#include <ctype.h>
#include <stdio.h>

#define IS_END_OF_SENTENCE(c)  ((c)=='.' || (c)=='?' || (c)=='!')

main()
{
    char c;
    int debut_phrase = 1;

    while ((c = getchar()) != EOF)
    {
        if (debut_phrase)
        {
            if (islower(c))
                putchar(toupper(c));
            else
                putchar(c);
            debut_phrase = !isalpha(c);
        }
        else
            if (isupper(c))
                putchar(tolower(c));
            else
            {
                putchar(c);
                debut_phrase = IS_END_OF_SENTENCE(c);
            }
```

```
    }
}
```
majuscule.c

Dans la session suivante, la commande est exécuté avec une redirection de son entrée standard depuis le fichier `MaJuScUlE` qu'on liste auparavant au moyen de la commande **CAT(1)**.

```
$ cat MaJuScUlE
dans un TEXTE. correcteMent TYpogrAPhie, les Phrases Commencent
pAr uNE Majuscule. ce texte est-il  CORRECTEMENT  tYpOgRaPhIe?
$
$ majuscule < MaJuScUlE
Dans un texte correctement typographie, les phrases commencent
par une majuscule. Ce texte est-il  correctement  typographie?
$
```

7.2.2 Valeur d'une chaîne numérique

`<stdlib.h>`

Plusieurs fonctions permettent de calculer la valeur d'une chaîne numérique (c'est-à-dire d'une chaîne dont les caractères sont des chiffres). Les trois plus simples sont :

- `double atof(const char *)` : conversion en réel flottant,
- `int atoi(const char *)` : conversion en entier standard,
- `long atol(const char *)` : conversion en entier long.

Voici l'exemple d'une calculatrice élémentaire évaluant une expression passée en argument. Les valeurs des opérandes sont calculées au moyen de la fonction `atof`.

calcul.c

```
#include <stdlib.h>
#include <stdio.h>
void usage(char *);

main(int argc, char *argv[])
{
    float v1, v2;

    if (argc != 4)
        usage(argv[0]);
    v1 = atof(argv[1]);
    v2 = atof(argv[3]);
    switch (argv[2][0])
    {
      case '+' :
        printf(" %g + %g = %g\n", v1, v2, v1+v2);
        break;

      case '-' :
```

```
            printf(" %g - %g = %g\n", v1, v2, v1-v2);
            break;

          case '*' :
            printf(" %g * %g = %g\n", v1, v2, v1*v2);
            break;

          case '/' :
            if (v2 == 0)
                printf(" Division par zero\n");
            else
                printf(" %g / %g = %g\n", v1, v2, v1/v2);
            break;

         default :
            usage(argv[0]);
      }
}

void usage(char *s)
{
    fprintf(stderr, "Usage: %s <n1> {+-*/} <n2>\n", s);
    exit(1);
}
```

calcul.c

Voici quelques exemples d'utilisation de cette commande :

```
$ calcul 127 / -3.5
 127 / -3.5 = -36.2857
$ calcul 256 '*' 17
 256 * 17 = 4352
$
```

Il existe égalcment des fonctions de conversion plus générales : `strtod` et `strtol`.

7.2.3 Chaînes de caractères

`<string.h> <stddef.h>`

Les chaînes de caractères sont étudiées dans la section 6.2. Nous présentons ici plus en détail les fonctions de la bibliothèque `string`, réalisant les opérations de recopie, des comparaison et des positionnement. Les fonctions de copie et de comparaison existent sous deux formes différentes. Avec les fonctions de la première forme,

```
char *strcpy(char *⟨destination⟩, const char *⟨source⟩)
char *strcat(char *⟨destination⟩, const char *⟨source⟩)
int strcmp(const char *⟨chaîne1⟩, const char *⟨chaîne2⟩)
```

la copie s'effectue jusqu'à la fin de la chaîne ⟨*source*⟩ et la comparaison jusqu'à la fin des deux chaînes si elles sont égales ou jusqu'à la première différence rencontrée. Avec celles de la seconde forme,

```
char *strncpy(char *⟨destination⟩,
              const char *⟨source⟩, size_t ⟨longueur⟩)
char *strncat(char *⟨destination⟩,
              const char *⟨source⟩, size_t ⟨longueur⟩)
int   strncmp(const char *⟨chaîne1⟩,
              const char *⟨chaîne2⟩, size_t ⟨longueur⟩)
```

le nombre de caractères à recopier ou à comparer est borné par le paramètre ⟨*longueur*⟩. Le fichier en-tête `<stddef.h>` n'est nécessaire que lorsqu'on a besoin de déclarer une variable de type `size_t`.

Comparaison de chaînes

Le résultat de l'exécution de

`strcmp(⟨chaîne1⟩, ⟨chaîne2⟩)`

est :

< 0, si ⟨*chaîne*$_1$⟩ est inférieure à ⟨*chaîne*$_2$⟩ selon l'ordre lexicographique;
$= 0$, si ⟨*chaîne*$_1$⟩ et ⟨*chaîne*$_2$⟩ sont deux chaînes identiques;
> 0, si ⟨*chaîne*$_1$⟩ est supérieure à ⟨*chaîne*$_2$⟩ selon l'ordre lexicographique.

Remarque 26 *Sous certaines implémentations non standard, les fonctions de bibliothèque* `strncpy` *et* `strncat` *n'ont pas des comportements homogènes. Si la longueur de la chaîne* ⟨*source*⟩ *est supérieure au paramètre* ⟨*longueur*⟩, *il est possible que la fonction* `strncpy` *ne rajoute pas le caractère* `'\0'` *après le dernier caractère copié. Par contre, la fonction* `strncat` *le rajoute dans tous les cas. Dans une implémentation standard, il est systématiquement recopié.*

La fonction `chaine_trier` qui suit est un exemple de tri d'un vecteur de chaînes de caractères utilisant la fonction de comparaison `strcmp` : deux mots sont échangés si le précédent est supérieur au suivant.

```
chaine_trier.c
#include <string.h>
#include "chaine_trier.h"

void chaine_trier(char *liste[], int taille_liste)
{
    int fin, courant;

    for (fin = taille_liste - 1; fin > 0; fin--)
    {
        int modifie = 0;

        for (courant = 0; courant < fin; courant++)
        {
            if (strcmp(liste[courant], liste[courant+1]) > 0)
            {
                /* Echange de courant et courant+1 */
                char *tmp = liste[courant];

                liste[courant] = liste[courant+1];
                liste[courant+1] = tmp;
```

```
                modifie = 1;
            }
        }
        if (!modifie)
            break;
    }
}
```

══════════════════ chaine_trier.c ══════

Le programme suivant illustre l'utilisation de cette fonction :

═════ test_chaine_trier.c ══════════════════

```
#include <stdio.h>

#include "chaine_trier.h"

char *mots[] =
{
    "Zero", "Un", "Deux", "Trois", "Quatre",
    "Cinq", "Six", "Sept", "Huit", "Neuf"
};
#define NBMOTS    (sizeof mots / sizeof mots[0])

main()
{
    int i;

    chaine_trier(mots, NBMOTS);
    for (i=0; i<NBMOTS; i++)
        printf("%s\n", mots[i]);
}
```

══════════════════ test_chaine_trier.c ══════

Le vecteur `mots` une fois trié est affiché sur la sortie standard.

```
$ test_chaine_trier
Cinq
Deux
Huit
Neuf
Quatre
Sept
Six
Trois
Un
Zero
$
```

Copie de chaînes

Il existe deux formes de copie : la recopie d'une chaîne à partir d'une adresse donnée (vecteur de caractères, autre chaîne...), et la concaténation d'une chaîne à la fin d'une autre chaîne. Dans chacun des deux cas, la chaîne source

est le second paramètre.

La fonction `strcpy` recopie la chaîne source à l'adresse spécifiée par son premier paramètre. La fonction `strcat` recherche la fin de la chaîne ayant pour adresse la valeur du premier paramètre formel, puis effectue la recopie de la chaîne source.

Ces fonctions sont de type `char *`, et retournent l'adresse du début de la chaîne résultat (c'est-à-dire le premier paramètre). Voici un exemple de copie et de concaténation de chaînes de caractères; il s'agit de la fonction `il_est_environ` qui traduit en français une heure exprimée numériquement. L'heure est approximée à dix minutes près.

```
═══════ il_est_environ.c ═══════════════════════════════
#include <string.h>
#include "il_est_environ.h"
#include "chaine.h"

static char *heures[] =
{
    "", "une", "deux", "trois", "quatre", "cinq",
    "six", "sept", "huit", "neuf", "dix", "onze"
};

static char *minutes[] =
{
    "", "dix", "vingt", "et demie",
    "moins vingt", "moins dix"
};

static const int precision=(sizeof minutes/sizeof minutes[0]);

char *il_est_environ(int h, int m)
{
    char buffer[64];
    char *demi_journee = NULL;

    if (m >= 30 + (30/precision))
        h++;
    /* Arrondi des minutes */
    m = ((precision*m + 30)/60) % precision;

    if (h == 12)
        strcpy(buffer, "midi");
    else if (h == 0 || h == 24)
        strcpy(buffer, "minuit");
    else
    {
        strcpy(buffer, heures[h % 12]);
        strcat(buffer, " heure");
        if (h % 12 > 1)
            strcat(buffer, "s");
        if (h < 12)
            demi_journee = " du matin";
        else if (h < 18)
            demi_journee = " de l'apres-midi";
        else
            demi_journee = " du soir";
```

```
    }
    if (m != 0)
        strcat(strcat(buffer, " "), minutes[m]);
    if (demi_journee != NULL)
            strcat(buffer, demi_journee);
    return chaine_dup(buffer);
}
```
il_est_environ.c

On remarquera l'expression

```
strcat(strcat(buffer, " "), minutes[m])
```

qui compose deux concaténations. La commande `heure` suivante permet de tester cette fonction.

heure.c

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

static void usage(char *);

main(int argc, char *argv[])
{
    if (argc != 3 )
        usage(argv[0]);

    if (argc == 3)
        printf("Il est environ %s.\n",
               il_est_environ(atoi(argv[1]), atoi(argv[2])));
}

static void usage(char *s)
{
    fprintf(stderr, "Usage: %s h m\n", s);
    exit(1);
}
```
heure.c

Elle s'utilise avec deux valeurs numériques en arguments, la première correspondant aux heures et la seconde aux minutes. Voici quelques exemples d'utilisation de cette commande :

```
$ heure 0 7
Il est environ minuit dix.
$ heure 11 34
Il est environ onze heures et demie du matin.
$ heure 11 35
Il est environ midi moins vingt.
$ heure 12 47
Il est environ une heure moins dix de l'apres-midi.
$ heure 22 55
Il est environ onze heures du soir.
$
```

Pour effectuer la concaténation, les fonctions `strcat` et `strncat` parcourent la chaîne transmise comme premier paramètre, afin de déterminer l'adresse à laquelle le second paramètre doit être recopié. Cela n'est pas très bien adapté à l'exemple précédent. Une amélioration consiste à définir une variante de la fonction de recopie retournant l'adresse du caractère `NULL` de la chaîne construite. De cette façon, il est possible d'enchaîner plusieurs appels à cette fonction sans provoquer de parcours redondants de la chaîne.

Longueur et positionnement

La fonction `strlen` retourne la longueur de la chaîne passée en paramètre. La longueur est le nombre de caractères *visibles* de la chaîne; le marqueur de fin de chaîne, le caractère `'\0'` n'est pas compté. Par conséquent, une chaîne ⟨*chn*⟩ occupe en réalité

`strlen(`⟨*chn*⟩`) + 1`

octets en mémoire. Il est possible de rechercher un caractère dans une chaîne au moyen des fonctions

`char *strchr(const char *`⟨*chaîne*⟩`, int `⟨*caractère*⟩`)`
`char *strrchr(const char *`⟨*chaîne*⟩`, int `⟨*caractère*⟩`)`

Elles retournent un pointeur vers respectivement la première et la dernière occurrence du caractère ⟨*caractère*⟩ dans la chaîne s'il est présent, et le pointeur `NULL` sinon.

Nous allons traiter l'exemple des fonctions `basename` et `dirname`, extrayant d'un chemin de fichier UNIX, respectivement le nom du fichier, et le répertoire contenant ce fichier. Par exemple, `dirname("/usr/local/bin/emacs")` retourne la chaîne `"/usr/local/bin"`, et la fonction `basename` retourne la chaîne `"emacs"`. Ces deux fonctions utilisent la fonction `strrchr` pour rechercher la dernière occurrence du caractère `/` dans le chemin.

Les deux fonctions `basename` et `dirname` allouent dynamiquement une zone mémoire, y recopient la partie du chemin à extraire et retournent l'adresse de la nouvelle chaîne ainsi créée.

```
basename.c
#include <stdlib.h>
#include <string.h>

char *basename(char *path)
{
    int l = strlen(path);
    char *p = (char *) strrchr(path, '/');

    if (p == 0)
        return strcpy(malloc(l+1), path);
    return strcpy(malloc(l-(p-path)), p+1);
}

char *dirname(char *path)
{
    int l = strlen(path);
    char *p = (char *) strrchr(path, '/');
```

```
    if (p == 0)
        return "";
    if (p == path)
        return "/";
    return strncpy(malloc(p-path+1), path, p-path);
}
```
basename.c

La fonction **basename** peut être utilisée pour extraire le nom de la commande à partir de l'argument argv[0] de la commande, par exemple lors de l'invocation de la fonction **usage** mentionnée en 4.3.3 :

```
usage(basename(argv[0]))
```

Le programme nomcmd suivant met en évidence l'intérêt de cette construction :

nomcmd.c

```
#include <stdio.h>

void usage(char *);
char *basename(char *);

main(int argc, char *argv[])
{
    if (argc == 1)
        usage(basename(argv[0]));
    printf("Execution de \"%s %s\"\n", argv[0], argv[1]);
}

void usage(char *name)
{
    printf("Usage: %s <argument>\n", name);
    exit (1);
}
```
nomcmd.c

Quelle que soit la manière dont est lancée la commande, la fonction **usage** affiche seulement le nom de la commande dans le message d'erreur :

```
$ nomcmd
Usage: nomcmd <argument>
$
$ nomcmd -v
Execution de "nomcmd -v"
$
$ /usr/labri/achille/Livre/2/src/nomcmd -v
Execution de "/usr/labri/achille/Livre/2/src/nomcmd -v"
$
$ /usr/labri/achille/Livre/2/src/nomcmd
Usage: nomcmd <argument>
$
```

Remarque 27 *Avant l'établissement de la norme* ANSI*, les fonctions* `strchr` *et* `strrchr` *existaient, dans plusieurs implémentations d'*UNIX*, sous les noms* `index` *et* `rindex`*. Cela peut créer des difficultés pour porter une application dans un environnement ne disposant que d'une implémentation non standard de la bibliothèque avec les formes obsolètes* `index` *et* `rindex`*. Il suffit en général dans ce cas de placer en tête des fichiers concernés les directives*

```
#define strchr index
#define strrchr rindex
```

7.3 Allocation dynamique de mémoire

<stdlib> <stddef>

(Le fichier `<stddef.h>` est seulement nécessaire pour disposer du type `size_t`.)

L'**allocation dynamique de mémoire** consiste à étendre, pendant l'exécution d'un programme, la mémoire qui lui est attribuée. Sous UNIX, dans un processus en exécution, les instructions d'une part, et les données d'autre part, sont situées dans des zones mémoire différentes. La zone mémoire contenant les instructions, encore appelée **région texte**, est chargée au lancement du programme et reste inchangée jusqu'à sa terminaison. La zone mémoire consacrée aux données peut par contre être agrandie en cours d'exécution.

Les fonctions d'allocation dynamique de la bibliothèque standard sont :

- `malloc` et `calloc` pour allouer un bloc mémoire;
- `realloc` pour étendre ou réduire sa taille;
- `free` pour le restituer.

Les fonctions d'allocation retournent l'adresse d'une zone mémoire continue; elles reçoivent en paramètre la taille en octets de la zone à allouer. Le prototype de `malloc` est

`void *malloc(size_t` ⟨*nombre-d'octets*⟩`)`

Pour allouer dynamiquement une donnée d'un type t, on calcule la taille en octets d'un objet de ce type au moyen de l'opérateur `sizeof`. Par exemple, si `struct cellule` est un identificateur de type, on pourra écrire

```
struct cellule *premiere = malloc(sizeof(struct cellule))
```

pour déclarer et initialiser un pointeur vers un objet de type `struct cellule`. Ce mécanisme peut être généricisé au moyen d'une pseudo-fonction :

```
#define ALLOC(t)     malloc(sizeof(t))
```

La fonction `calloc` s'emploie avec deux paramètres :

`void *calloc(size_t` ⟨*nombre-elem*⟩`, size_t` ⟨*taille-elem*⟩`)`

Elle retourne l'adresse d'une zone de taille *nombre-elem* × *taille-elem* octets. De plus, la mémoire allouée est initialisée avec des zéros. Si cette initialisation est inutile, on pourra utiliser la pseudo-fonction

```
#define CALLOC(n, t)     malloc((n) * sizeof(t))
```

La fonction `realloc` reçoit en paramètre l'adresse d'un bloc mémoire à étendre et sa nouvelle taille :

void *realloc(void *⟨*adresse*⟩, size_t ⟨*nouvelle-taille*⟩)

Elle alloue un nouveau bloc de taille ⟨*nouvelle-taille*⟩ et recopie dedans le contenu de l'ancien. Ce dernier est ensuite libéré.

La fonction `free` libère le bloc mémoire précédemment alloué par une fonction d'allocation. Elle reçoit l'adresse du bloc en paramètre :

void free(void *⟨*adresse*⟩)

Nous allons utiliser ces outils pour écrire une fonction lisant une chaîne de caractères de longueur quelconque. La chaîne est lue dans une mémoire intermédiaire, d'une taille initiale TBLOC; cette zone est étendue au moyen de la fonction `realloc` chaque fois que le nombre de caractères lus dépasse sa taille. Lors de la lecture, le caractère \ en fin de ligne joue le rôle de caractère de continuation. La taille TBLOC a été fixée à huit octets pour les besoins du test.

Lorsque la lecture est terminée, la fonction alloue une zone mémoire de la taille de la chaîne lue, y recopie son contenu, et libère la mémoire intermédiaire au moyen de la fonction `free`. Elle retourne enfin l'adresse de la chaîne.

lire_chaine.c

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

#include "ERREURS.h"
#include "exit_if.h"

#define TBLOC   8

char *lire_chaine(char *prompt)
{
    int  taille = TBLOC;
    char *buffer = malloc(taille);
    char *p = buffer;

    EXIT_IF(buffer == NULL, ERR_MALLOC);

    printf("%s", prompt);
    /* Boucle de lecture */
    for (;;)
    {
        *p = getchar();
        if (*p == EOF)
        {
            free(buffer);
            return NULL;
        }
        if (*p == '\n')
        {
            if (p > buffer && *(p-1) == '\\')
            {
                /* annulation du \n (precede de \) */
                p--;
                continue;
            }
            /* fin de la saisie */
```

```
            break;
        }
        if (++p == buffer + taille)
        {
            /* Agrandissement de la memoire de lecture */
            buffer = realloc(buffer, taille+TBLOC);
            EXIT_IF(buffer == NULL, ERR_REALLOC);
            p = buffer + taille;
            taille += TBLOC;
        }
    }
    *p = '\0';

    /* Allocation d'une zone de la taille de la chaine */
    p = malloc(p - buffer + 1);
    EXIT_IF(p == NULL, ERR_MALLOC);
    strcpy(p, buffer);

    /* Liberation de la memoire intermediaire */
    free(buffer);
    return p;
}
```
lire_chaine.c

Il est possible que le processus atteigne la taille maximum de la mémoire dont il dispose en machine. Dans ce cas, les appels à `malloc` ou à `realloc` peuvent échouer; ils retournent alors le pointeur `NULL`, défini dans `<stddef.h>`. Dans ce cas, la pseudo-fonction `EXIT_IF` provoque une terminaison. Les messages d'erreurs passés en paramètre à `EXIT_IF` sont définis dans `ERREURS.h`

Le programme suivant est un exemple d'utilisation de cette fonction. Il itère des appels à `lire_chaine` jusqu'à la lecture d'une chaîne vide.

test_lirechaine.c

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

#include "lire_chaine.h"

main()
{
    for(;;)
    {
        char *s;

        s = lire_chaine("Chaine: ");
        if (s == NULL)
            break;
        printf("  lu \"%s\" de longueur %d\n", s, strlen(s));
        free(s);
    }
    printf("\nFin du test.\n");
}
```
test_lirechaine.c

Il est possible de lire une chaîne arbitrairement longue et de continuer sur la ligne suivante en terminant la ligne courante par un \.

```
$ test_lirechaine
Chaine: a\bc\d
  lu "a\bc\d" de longueur 6
Chaine: abcdef\
ghijklmn\
opqrstuvwxyz
  lu "abcdefghijklmnopqrstuvwxyz" de longueur 26
Chaine:
  lu "" de longueur 0
Chaine: \
\
x
  lu "x" de longueur 1
Chaine: C-d
Fin du test.
$
```

Nous allons considérer un second exemple de construction d'une chaîne de longueur quelconque par mécanisme d'allocation/rallongement. Il s'agit d'une fonction extrayant les options de la liste des arguments d'une commande. Les options sont identifiées par un tiret suivi d'un ou plusieurs caractères. Par exemple, la liste de mots

`-c -vf out f1 -v`

contient quatre options : `c`, `v`, `f` et à nouveau `v`. La fonction `extraire_options` s'utilise avec deux paramètres :

- un vecteur de chaînes de caractères codant la liste des arguments d'une commande (`argv` par exemple);
- une chaîne de caractères contenant la liste des options autorisées.

Un tiret seul provoque la fin de la recherche des options dans la liste d'arguments. La fonction retourne une chaîne contenant la liste des options reconnues si elles sont toutes valides (`"c v f v"` sur l'exemple précédent) et `NULL` sinon.

```
═══════ options.c ═══════════════════════
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <stddef.h>

#include "ERREURS.h"
#include "exit_if.h"

#include "options.h"

#define NBOPT          32
#define CARACTERE_OPT  '-'

char *extraire_options(char **args, char *options_valides)
```

```
{
   int  taille_opt = NBOPT+1;
   char *liste_opt = malloc(taille_opt);
   int  ind_option = 0;

   EXIT_IF(liste_opt==0, ERR_MALLOC);

   for (; *args != NULL; args++)
   {
      char *p = *args;

      if (*p != CARACTERE_OPT)
          continue;
      if (*++p == '\0')
          return liste_opt;
      while (*p != '\0')
      {
         if (strchr(options_valides, *p) == NULL)
         {
            free(liste_opt);
            return NULL;
         }
         if (ind_option == taille_opt)
         {
            taille_opt += NBOPT;
            liste_opt = realloc(liste_opt, taille_opt);
            EXIT_IF(liste_opt == NULL, ERR_MALLOC);
         }
         liste_opt[ind_option++] = *p;
         p++;
      }
   }
   return liste_opt;
}
```
options.c

Le programme `extropts.c` est un exemple d'utilisation de cette fonction. Il s'agit d'une commande s'utilisant de la façon suivante :

extropts ⟨*options*⟩ ⟨*arg*⟩...

et affichant sur la sortie standard la liste des options extraites.

extropts.c
```
#include <stdio.h>
#include <stddef.h>

#include "options.h"

main(int argc, char **argv)
{
    char *opts;
    char *p;

    if (argc < 3)
    {
        fprintf(stderr, "Usage: %s <options> <arg>...\n",argv[0]);
        exit(1);
```

```
    }
    opts = extraire_options(argv+2, argv[1]);
    if (opts == NULL)
        exit(1);
    for (p = opts; *p != '\0'; p++)
        printf("%c ", *p);
    printf("\n");
    free(opts);
    exit(0);
}
```

extropts.c

Cette commande est utilisable depuis une commande shell; la valeur de terminaison (voir 12.1.4) permet de déterminer si toutes les options sont valides.

```
$ extropts crRvf -cv -f tmp -R
c v f R
$ extropts crRvf -R -c tmp1 -f tmp2
R c f
$ extropts crRvf -R - -xyz
R
$ extropts crRvf -R -x
$
```

Il peut sembler excessif de mettre en œuvre un mécanisme d'allocation/rallongement pour construire une chaine dont la longueur ne dépasse pas à priori quelques caractères, mais c'est la condition *sine qua non* à la réutilisabilité de la fonction et à la non limitation de la commande.

Remarque 28 *Les blocs retournés par les fonctions d'allocation mémoire sont en général alignés sur des frontières de mots. De plus, un en-tête contenant la taille de la zone allouée est placé en début de bloc (l'adresse retournée est l'adresse du premier mot suivant l'en-tête).*

Cette remarque entraine deux conséquences :

1) la taille de la zone allouée est supérieure à celle demandée par l'utilisateur; une allocation dynamique octet par octet peut s'avérer coûteuse;
2) il est impératif que les adresses passées en paramètres à `realloc` ou à `free` soient des adresses retournées par de précédents appels à une fonction d'allocation mémoire afin que le bloc mémoire ait un en-tête valide.

Le programme suivant illustre la première conséquence. La fonction `malldmp` imprime l'adresse d'un bloc mémoire et la valeur de son en-tête, c'est-à-dire sa taille réelle. Attention, ce programme dépend de l'implémentation de la fonction `malloc` et n'est pas portable. En particulier, il ne respecte pas le principe énoncé page 179.

malldmp.c

```
#include "malldmp.h"

#define NB_MOTS_ENTETE 2

void malldmp(void *s)
```

```
{
    printf("0x%x %d (%d)", s, s, *((int *) s-NB_MOTS_ENTETE));
}
```

malldmp.c

Le programme test_malldmp enchaîne plusieurs allocations en affichant à chaque fois :

- le nombre d'octets demandé,
- l'adresse en hexadécimal et en décimal retournée par `malloc`,
- la taille réelle du bloc.

On peut voir que dans cette implémentation, une demande d'allocation d'une zone d'un octet se traduit par l'allocation de seize octets : 1 octet demandé + 8 octets pour l'en-tête + 7 octets pour se ramener à une frontière de double mot (ici un multiple de huit). On peut également vérifier que la différence entre les adresses de deux blocs i et $i+1$ correspond à la taille du bloc i.

```
$ malldmp
malloc(01) : 0x6198 24984 (16)
malloc(02) : 0x61a8 25000 (16)
malloc(03) : 0x61b8 25016 (16)
malloc(04) : 0x61c8 25032 (16)
malloc(05) : 0x61d8 25048 (16)
malloc(06) : 0x61e8 25064 (16)
malloc(07) : 0x61f8 25080 (16)
malloc(08) : 0x6208 25096 (16)
malloc(09) : 0x6218 25112 (24)
malloc(10) : 0x6230 25136 (24)
malloc(11) : 0x6248 25160 (24)
malloc(12) : 0x6260 25184 (24)
malloc(13) : 0x6278 25208 (24)
malloc(14) : 0x6290 25232 (24)
malloc(15) : 0x62a8 25256 (24)
malloc(16) : 0x62c0 25280 (24)
malloc(17) : 0x62d8 25304 (32)
$
```

Voici maintenant un exemple illustrant la seconde conséquence de la remarque; il s'agit d'un programme contenant l'erreur consistant à appeler la fonction `realloc` avec une adresse invalide.

erralloc.c

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

main()
{
    char *p = malloc(1);

    printf("Adresse initiale: %x\n", (int) p);
    p = realloc(p, 256);
    printf("- premiere extension : %x\n", (int) p);
```

```
    p++;
    p = realloc(p, 1024);
    printf("- seconde extension : %x\n", (int) p);
}
========================================== erralloc.c ==========
```

Le second appel à `realloc` retourne une erreur car l'adresse passée en paramètre est incorrecte (elle n'est plus alignée sur une frontière de mot).

```
$ erralloc
Adresse initiale: 231a8
- premiere extension : 2323c
- seconde extension : 0
$
```

La fonction suivante est un exemple d'uniformisation des fonctions d'allocation et de rallongement. Elle reçoit en paramètre un pointeur et un nombre d'octets, et retourne l'adresse de la zone allouée. Si le pointeur est nul, elle effectue une allocation, sinon un rallongement.

```
========== allocation.c ==========================================
#include <stdlib.h>
#include <stddef.h>

char *allocation(char *adresse, int taille)
{

    if (taille == 0)
        taille = 1;

    if (adresse == NULL)
        adresse = malloc(taille);
    else
        adresse =  realloc(adresse, taille);
    return adresse;
}
========================================== allocation.c ==========
```

Le programme suivant est une illustration de l'utilisation de cette fonction :

– allocation d'un bloc mémoire initial pour copier la chaîne `"- Debut -"`;

– extension de ce bloc pour concaténer la chaîne `"- Suite -"`.

```
========== dmpchn.c ==============================================
#include <stdio.h>
#include <stddef.h>
#include <string.h>

#include "malldmp.h"
#include "allocation.h"

static void dmpchn(char *s);
```

```
char *chaine1 = "- Debut -";
char *chaine2 = " Milieu - Fin -";

main()
{
    char *p = NULL;

    p = allocation(p, strlen(chaine1)+1);
    strcpy(p, chaine1);
    dmpchn(p);
    p = allocation(p, strlen(chaine1)+strlen(chaine2)+1);
    strcat(p., chaine2);
    dmpchn(p);
}

static void dmpchn(char *s)
{
    printf(" Impl. ");
    malldmp(s);
    printf(" --> \"%s\" (%d car.)\n", s, strlen(s)+1);
}
```
dmpchn.c

L'exécution de ce programme produit un affichage de la forme :

```
 Impl. 0x6140 24896 (24) --> "- Debut -" (10 car.)
 Impl. 0x61e8 25064 (40) --> "- Debut - Milieu - Fin -" (25 car.)
$
```

7.4 Assertions et mise au point de programme

<assert.h>

Il existe un mécanisme rudimentaire de vérification d'assertions. Il est implémenté au moyen de la pseudo-fonction `assert` située dans le fichier en-tête `<assert.h>`. Celle-ci est appelée avec en paramètre une expression à évaluer, et provoque une terminaison par un appel à la fonction `abort`, lorsque l'expression est fausse (c'est-à-dire de valeur nulle). Elle imprime la localisation de l'assertion fausse (nom du fichier et numéro de ligne).

Nous allons voir, pour illustrer ce mécanisme, un exemple de mise au point d'une fonction, de nom `itob`, construisant la chaîne de zéros et de uns de la représentation en base 2 d'un entier.

Cette fonction contient deux assertions, rajoutées dans la boucle afin d'aider à la détection d'une erreur se produisant lors de l'exécution du programme.

itob.c

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

```
#define LONGUEUR_MOT 32
static char *itob(unsigned);

main(int argc, char *argv[])
{
    if (argc != 2)
        exit(1);
    printf("%d s'ecrit %s en base 2\n", atoi(argv[1]),
           itob((unsigned) atoi(argv[1])));
}

static char *itob(unsigned i)
{
    static char b[LONGUEUR_MOT+1];
    char *pb = & b[LONGUEUR_MOT];

    *pb = '\0';
    do
    {
        assert(pb > b);
        *--pb = (i & 1);
        assert(*pb=='0' || *pb=='1');
        i >>= 1;
    }
    while (i);
    return pb;
}
```

itob.c

Le programme est compilé avec l'option de mise au point `-g` afin que le compilateur génère les informations nécessaires au débogueur symbolique (ici le débogueur `gdb` du projet *GNU*). C'est d'ailleurs une bonne habitude que de positionner cette option systématiquement. Au cours de l'exécution de `itob 126`, la seconde assertion se révèle fausse.

```
$ gcc -g itob.c -o itob
$
$ itob 126
itob.c:26: failed assertion '*pb=='0' || *pb=='1''
Abort (core dumped)
$
```

On lance le débogueur symbolique pour rechercher l'erreur. La commande `l 26` fait lister quelques lignes entourant la ligne numéro 26, sur laquelle l'assertion a échoué.

```
$ gdb itob
GDB is free software and you are welcome to distribute copies
of it under certain conditions;  type "show copying"  to  see
the conditions. There is absolutely no warranty for GDB; type
"show warranty" for details.
GDB 4.8, Copyright 1993 Free Software Foundation, Inc...
(gdb)
(gdb) l 26
21          *pb = '\0';
```

```
22          do
23          {
24              assert(pb > b);
25              *--pb = (i & 1);
26              assert(*pb=='0' || *pb=='1');
27              i >>= 1;
28          } while (i);
29          return pb;
30      }
(gdb)
```

On place un point d'arrêt sur la ligne 26 au moyen de la commande `b 26` (`b` est une abréviation de `breakpoint`), et on relance le programme avec `126` en argument en tapant `run 126`:

```
(gdb) b 26
Breakpoint 1 at 0x2404: file itob.c, line 26.
(gdb) run 126
Starting program:/usr/labri/achille/Livre/2/src/itob 126
Breakpoint 1, itob (i=126) at itob.c:26
26              assert(*pb=='0' || *pb=='1');
(gdb)
```

Le programme s'interrompt lorsqu'il arrive sur l'instruction de la ligne 26 en l'affichant à l'écran. Il est alors possible de visualiser la valeur des différentes variables. On utilise pour cela la commande `p` (abréviation de `print`) :

```
(gdb) p i
$1 = 126
(gdb) p *pb
$2 = 0 '\000'
(gdb)
```

Le caractère courant `*pb` devrait être le caractère `0`; au lieu de cela, il contient le caractère `\0`, de code ASCII 0. L'erreur se situe dans l'expression

```
*--pb = (i & 1)
```

l'expression correcte est

```
*--pb = (i & 1) + '0'
```

On peut terminer la session de débogage et corriger l'erreur.

```
(gdb) quit
The program is running.  Quit anyway? (y or n) y
$
```

Le programme corrigé s'exécute normalement :

```
$ itob 126
126 s'ecrit 1111110 en base 2
$
```

Il est possible de désactiver le mécanisme d'assertion lorsque le programme est supposé correct. Il suffit pour cela de compiler le programme avec l'option `-DNDEBUG`.

Le debogeur symbolique `gdb` offre un grand nombre de possibilités. Voici, à titre d'exemple, une autre session utilisant la version corrigée du programme précédent. On place, tout d'abord, un point d'arrêt sur la première instruction du programme; on lance ensuite son exécution sans argument :

```
(gdb) b main
Breakpoint 1 at 0x22c0: file itob.c, line 9.
(gdb) run
Starting program: /usr/labri/achille/Livre/2/src/itob
Breakpoint 1, main (argc=1, argv=0xf7fff974) at itob.c:9
9           if (argc != 2)
(gdb)
```

Les lignes suivantes montrent comment effectuer des modifications de variables du programme, ici `argc` et `argv`, directement depuis le débogueur, au moyen de la commande `set`.

```
(gdb) p argc
$1 = 1
(gdb) set argc=2
(gdb) set argv[1] = "10"
(gdb) p argc
$7 = 2
(gdb) p argv[1]
$5 = 0x63d0 "10"
(gdb)
```

On peut également faire exécuter le programme ligne par ligne, en *pas à pas*:

```
(gdb) s
11          printf("%d s'ecrit %s en base 2\n",
(gdb)
```

Le débogueur imprime, à chaque fois, le source de la prochaine ligne à exécuter. Il est possible d'évaluer n'importe quelle expression C correcte, y compris une expression effectuant des appels de fonction:

```
(gdb) p atoi(argv[1])
$9 = 10
(gdb)
```

On continue l'exécution pas à pas, en faisant éventuellement imprimer des valeurs de variables. La frappe d'une ligne vide provoque une nouvelle exécution de la dernière commande exécutée.

```
(gdb) s
itob (i=10) at itob.c~:19
19          char *pb = & b[32];
(gdb)
21          *pb = '\0';
```

```
(gdb)
24              assert(pb > b);
(gdb)
25              *--pb = (i & 1) + '0';;
(gdb)
26              assert(*pb=='0' || *pb=='1');
(gdb)
27              i >>= 1;
(gdb) p i
$10 = 10
(gdb) s
28          } while (i);
(gdb) p i
$11 = 5
(gdb)
```

Cette session ne présente qu'une petite partie des possibilités du débogueur **gdb**. La figure 7.2 illustre l'utilisation de **gdb** sous **emacs**. L'écran est séparé en deux fenêtres. Dans celle du haut, il est possible d'entrer les commandes pour **gdb**, et dans celle du bas, on est en permanence positionné sur la prochaine instruction à exécuter (celle-ci est indiquée par une flèche **=>**).

```
emacs-19.16@nemo
Buffers   File   Edit   Help
GDB 4.8, Copyright 1993 Free Software Foundation, Inc...
(gdb) b 26
Breakpoint 1 at 0x2404: file itob.c, line 26.
(gdb) run 123
Starting program: /a/firmin/export/home/achille/Livre/2/src/itob 123

Breakpoint 1, itob (i=123) at itob.c:26
(gdb) p i
$1 = 123
(gdb) p *pb
$2 = 1 '\001'
(gdb) p b
$3 = 0x4380 '\000' <repeats 31 times>, "\001"
(gdb) █
--**-Emacs: *gud-itob*        (Debugger: run)--Bot----------------------------
    char *pb = & b[LONGUEUR_MOT];

    *pb = '\0';
    do
    {
        assert(pb > b);
        *--pb = (i & 1);
=>      assert(*pb=='0' || *pb=='1');
        i >>= 1;
    }
    while (i);
-----Emacs: itob.c            (C)--83%--------------------------------------
Mark set
```

Figure 7.2: Utilisation de gdb sous emacs

De façon standard, l'implémentation *GNU* de `assert` produit une terminaison sur exception avec production d'un cliché mémoire (`core`), examinable au moyen du débogueur. Certains autres compilateurs génèrent une terminaison normale au moyen d'un appel à `exit`. Voici un exemple de redéfinition de la pseudo-fonction `assert` :

```
═══════ assert.h ════════════════════════
#ifndef ASSERT_H
#define ASSERT_H

#ifndef NDEBUG
#   include <stdio.h>

#   ifdef assert
#       undef assert
#   endif /* assert */

#   define assert(e)  {\
if (!(e)) \
{\
    fprintf(stderr,\
            "%s:ligne %d: assertion \"%s\" non verifiee\n",\
            __FILE__, __LINE__, #e);\
    abort();\
}}
#else
#   define assert(ignore)
#endif /* NDEBUG */

#endif  /* ASSERT_H */
═════════════════════════════ assert.h ═══════
```

Si on recompile le programme erroné précédent en remplaçant la ligne

```
#include <assert.h>
```

par

```
#include "assert.h"
```

on obtient lors de son exécution le message d'erreur :

```
$ assertion 126
itob.c:ligne 26: assertion "*pb=='0' || *pb=='1'" non verifiee
Abort (core dumped)
$
```

Cette définition de la pseudo-fonction `assert` utilise le mécanisme de génération de chaîne de caractères présenté en 5.1.4, page 158.

7.5 Accès à l'environnement utilisateur

`<stdlib.h>`

L'environnement utilisateur, ou plus simplement **environnement**, d'un programme est une liste de variables chaînes de caractères, c'est-à-dire de couples

$$\prec identificateur, chaîne \succ$$

Sous UNIX, parmi les variables couramment définies, on trouve

- TERM : type de terminal (*vt100*, xterm...) – ce nom est utilisé par les fonctions de bibliothèques vidéo comme termcap ou curses;
- USER, ou LOGNAME : nom de l'utilisateur;
- HOME : répertoire d'accueil;
- SHELL : nom de l'interprète de commandes;
- PATH : chemin des répertoires à parcourir pour rechercher les fichiers exécutables;
- DISPLAY : sous le système de gestion de fenêtres X Window, identification du serveur.

On peut obtenir la liste complète des variables de son environnement au moyen de la commande printenv. Il existe plusieurs façons d'accéder à la liste des variables d'environnement depuis un programme. Sous UNIX, elle est traditionnellement passée en paramètre à la fonction main, à la suite de argv. Pour obtenir cette liste, il suffit de déclarer la fonction main de la façon suivante :

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

Le parcours de cette liste peut être effectué de la manière suivante :

```
for (i=0; envp[i]; i++)
    faire quelque chose avec envp[i];
```

Chaque élément de la liste est de la forme

⟨*identificateur*⟩=⟨*chaîne*⟩

et la liste est terminée par un pointeur nul. Cette solution n'a pas été retenue dans la norme ANSI. Elle est cependant très utile sous UNIX pour créer un environnement à un processus que l'on crée (voir 12.3.1).

L'autre solution, conforme à la norme ANSI, utilise une fonction recherchant la valeur de la variable dont elle reçoit le nom en paramètre. Il s'agit de la fonction getenv, dont le prototype est

```
char *getenv(const char *⟨nom⟩)
```

Si la variable ⟨*nom*⟩ est définie, la fonction getenv retourne un pointeur vers une chaîne contenant sa valeur. Dans le cas contraire, elle retourne la valeur NULL. Attention, la chaîne retournée par getenv n'est pas modifiable.

Ce mécanisme est très utile pour paramétrer le comportement d'une application. L'exemple typique pour une application manipulant des fichiers est une variable contenant le chemin d'accès à ces fichiers. La mise en œuvre est la suivante :

racine.c

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

#define RACINE          "RACINE"
#define DEFAUT_RACINE   "."

main()
{
    char *racine;
```

```
    if ((racine = getenv(RACINE)) == 0)
        racine = DEFAUT_RACINE;
    printf("repertoire de travail: \"%s\"\n", racine);
}
```
racine.c

On peut ainsi modifier le comportement de cette commande, simplement en changeant la valeur de la variable RACINE de l'environnement shell.

```
$ racine
repertoire de travail: "."
$ export RACINE=/tmp
$
$ racine
repertoire de travail: "/tmp"
$
$ unset RACINE; racine
repertoire de travail: "."
$
```

7.6 Temps interne et temps externe

<time.h>

Les fonctions de manipulation du temps de la bibliothèque standard sont calquées sur celles du système UNIX. Le temps est exprimé sous une forme interne au système. Sous UNIX, il s'agit du nombre de secondes écoulées depuis le premier janvier 1970, 0 heure 0 minute et 0 seconde, heure de Greenwich. La fonction `time` permet d'obtenir cette valeur. Son prototype est

`time_t time(time_t *⟨temps⟩)`

Le résultat est transmis de deux façons différentes :

- comme valeur de retour de la fonction `time`;
- recopié dans le paramètre passé par référence ⟨*temps*⟩ si la valeur de ce paramètre est différente de NULL.

Plusieurs fonctions permettent de passer du **temps interne** au **temps externe**, encore appelé *temps local*, qui est une décomposition du temps interne en année, mois, jour, heure, etc. De même, il est possible de convertir un temps externe en temps interne.

La fonction

`struct tm* localtime(const time_t *⟨temps⟩)`

retourne la référence à une structure, `struct tm`, décrivant le temps externe. Le tableau 7.4 liste les champs de cette structure contenant les informations de date et temps. Ils sont tous de type entier,

La fonction

`char *ctime(const time_t *⟨temps⟩)`

`tm_year`	année - 1900
`tm_mon`	mois (de 0=janvier à 11=décembre)
`tm_wday`	jour de la semaine (de 0=dimanche à 6=samedi)
`tm_mday`	jour du mois (de 0 à 31)
`tm_yday`	jour de l'année (de 0 à 365)
`tm_hour`	heures (de 0 à 23)
`tm_min`	minutes (de 0 à 59)
`tm_sec`	secondes (de 0 à 59)

Tableau 7.4: Description du temps externe

construit une chaîne de caractères décrivant le temps externe dans un format américain standard, à partir du temps interne. La fonction

```
size_t strftime(char *⟨adresse⟩, size_t ⟨taille⟩,
                const char *⟨format⟩, const struct tm *⟨tl⟩)
```

effectue un formatage du temps externe ⟨*tl*⟩ dans la zone mémoire ⟨*adresse*⟩ sur une longueur d'au plus ⟨*taille*⟩ caractères. Le format est composé de spécifications analogues à celles utilisées par `printf` (voir tableau 7.5)

La fonction

```
double difftime(time_t *⟨temps1⟩, time_t *⟨temps2⟩)
```

calcule la différence en secondes entre deux valeurs de temps interne. La fonction

```
time_t mktime(struct tm* ⟨tl⟩)
```

convertit un temps externe en temps interne. Cette dernière fonction permet de calculer un intervalle de temps interne de façon portable, quelle que soit sa représentation. Par exemple, pour calculer le temps interne correspondant à un futur de cinq minutes, on peut écrire :

```
time_t temps_courant = time(NULL);
struct tm *temps_externe = localtime(&temps);
time_t temps_futur;

temps_externe.tm_min += 5;
temps_futur = mktime(temps_externe);
```

Voici un programme illustrant quelques uns de ces mécanismes.

```
time_f.c
#include <time.h>
#include <stdio.h>

#define TFORMAT   256

main()
{
    time_t temps = time(NULL);
    struct tm *temps_local = localtime(&temps);
    char   format[TFORMAT];
```

```
    strftime(format, sizeof format,
            "Il est %I:%M%p, soit %Hh %Mmn", temps_local);
    printf("\n%s\n", format);

    printf("- format standard: %s", ctime(&temps));

    strftime(format, sizeof format, "%x %X", temps_local);
    printf("- forme locale: %s\n", format);

    strftime(format, sizeof format, "%A %B %d %H:%M",temps_local);
    printf("- divers: %s\n", format);

}
```
time_f.c

La forme locale est spécifique à chaque implémentation.

```
$ time_f

Il est 11:52PM, soit 23h 52mn
- format standard: Wed Jun 23 23:52:28 1993
- forme locale: 06/23/93 23:52:28
- divers: Wednesday June 23 23:52
$
```

Voici un autre exemple d'utilisation de la fonction `localtime`. Il s'agit de la fonction `hms`, affectant à des paramètres passés par référence respectivement les heures, les minutes, et les secondes du temps courant.

hms.c

```
#include <time.h>

#include "hms.h"

void hms(int *heures, int *minutes, int *secondes)
{
    time_t   clock;
    struct tm *p;

    time(&clock);
    p = localtime(&clock);
    *heures   = p->tm_hour;
    *minutes  = p->tm_min;
    *secondes = p->tm_sec;
}
```
hms.c

Le programme suivant est un exemple d'utilisation de la fonction `hms`. Il utilise également la fonction `il_est_environ` définie en 7.2.3. La fonction `hms` sera utilisée plusieurs fois dans la suite de cet ouvrage.

%a	nom abrégé du jour de la semaine
%A	nom complet du jour de la semaine
%b	nom abrégé du mois
%B	nom complet du mois
%c	représentation locale date et heure
%d	jour du mois
%H	heure entre 00 et 23
%I	heure entre 00 et 12
%j	jour de l'année
%m	numéro du mois
%M	minutes
%p	notation pour *matin* et *après-midi*
%S	secondes
%U	numéro de la semaine (débutant un dimanche)
%w	numéro du jour
%W	numéro de la semaine (débutant un lundi)
%x	représentation locale de la date
%X	représentation locale de l'heure
%y	deux derniers chiffres de l'année
%Y	année complète
%Z	*time zone*

Tableau 7.5: Formats reconnus par la fonction `strftime`

test_hms.c

```
#include <stdio.h>

#include "hms.h"
#include "il_est_environ.h"

main()
{
    int   h, m, s;

    hms(&h, &m, &s);
    printf("Il est environ %s.\n", il_est_environ(h, m));
    printf("(%dh %dmn %ds, pour etre precis...)\n", h, m, s);
}
```

test_hms.c

```
$ date
Mon May 31 02:10:05 MET DST 1993
$
$ test_hms
Il est environ deux heures dix du matin.
(2h 10mn 7s, pour etre precis...)
$
```

7.7 Fonctions avec paramètres variables

<stdarg.h>

Il existait dans les implémentations UNIX du langage C un mécanisme permettant d'écrire des fonctions avec un nombre variable de paramètres. Ce mécanisme s'appuyait sur des pseudo-fonctions définies dans le fichier en-tête <varargs.h>. Il fait maintenant partie de l'environnement standard du langage C norme ANSI. Une des raisons de cet ajout est de permettre une implémentation standard de fonctions de bibliothèque comme sprintf ou scanf.

Une fonction avec paramètres variables doit avoir au moins un paramètre fixe. Cela sous-entend que le type de ce paramètre est prédéterminé. Il peut bien entendu y avoir plusieurs paramètres fixes. Les paramètres variables sont parcourus comme une liste, le parcours débutant à partir du dernier paramètre fixe.

Un appel à une fonction avec paramètres variables s'effectue comme un appel à n'importe quelle autre fonction. Considérons par exemple une fonction de nom concatener, recevant en paramètre un nombre quelconque de chaînes de caractères et retournant la concaténation de toutes ces chaînes dans une zone mémoire allouée dynamiquement. Les appels à cette fonction seront de la forme

```
concatener(ch1, ch2)
concatener("<<", argv[0], ">>")
concatener("(", s1, ", ", s2, ", ", s3, ")")
...
```

La récupération des paramètres s'effectue au moyen de quatre constructions définies dans le fichier en-tête standard <stdarg.h> :.

- va_list : déclaration de la structure de données qui sera utilisée pour effectuer le parcours de la liste;
- va_start : initialisation du parcours sur le premier paramètre variable;
- va_arg : obtention d'un paramètre et passage au suivant;
- va_end : terminaison du parcours.

La construction va_list a le statut syntaxique d'une déclaration. Elle doit par conséquent être placée avant la première instruction de la fonction. Elle prend en paramètre le nom de la structure de données utilisée pour le parcours. Reprenons l'exemple de la fonction concatener. Il est raisonnable qu'un appel à cette fonction comporte au moins un paramètre, que nous appellerons premiere_chaine. Le début de concatener est de la forme :

```
char *concatener(char *premiere_chaine, ...)
{
    va_list chaines;
```

La construction va_start a le statut syntaxique d'une fonction sans valeur de retour. Elle reçoit deux paramètres :

- la structure de données déclarée par va_list;

– le nom du paramètre formel à la suite duquel le parcours des paramètres doit commencer.

Sur notre exemple, cela donne :

```
va_start(chaines, premiere_chaine);
```

La construction `va_arg` a le statut syntaxique d'une fonction de type variable. Elle s'emploie avec deux paramètres :

– la structure de données déclarée par `va_list`,
– le type du paramètre suivant à récupérer dans la liste.

Elle retourne la valeur de ce paramètre. Sur notre exemple, tous les paramètres sont des chaînes de caractères. La récupération des paramètres peut donc s'écrire au moyen d'une boucle de la forme :

```
for (;;)
{
    char *p = va_arg(chaines, char *);
```

Il est nécessaire de choisir une convention pour détecter la fin de la liste, par exemple en terminant la liste par le pointeur `NULL`. Cela impose une contrainte lors de l'appel, mais celle-ci est inévitable, car aucun mécanisme automatique de détection de la fin de la liste n'est fourni. Les exemples d'appel que nous avons donnés plus haut sont donc incomplets et doivent être modifiés de la façon suivante :

```
concatener(ch1, ch2, NULL)
concatener("<<", argv[0], ">>", NULL)
concatener("(", s1, ", ", s2, ", ", s3, ")", NULL)
...
```

Dans notre exemple, la sortie de la boucle de récupération s'écrit :

```
if (p == NULL)
    break;
```

Une fois la récupération terminée, il est nécessaire d'appeler la construction `va_end`, avec en paramètre la structure de données utilisée pour effectuer le parcours :

```
va_end(chaines);
```

Nous sommes maintenant en mesure d'écrire l'exemple complet. Nous allons effectuer deux parcours de la liste des paramètres. Le premier sert à totaliser la longueur de toutes les chaînes reçues afin d'effectuer l'allocation dynamique de la chaîne recevant le résultat de la concaténation. Le second parcours effectue la concaténation.

concatener.c

```
#include <stdlib.h>
#include <string.h>
#include <stddef.h>
#include <stdarg.h>

#include "exit_if.h"
#include "ERREURS.h"
#include "concatener.h"
```

```
char *concatener(char *premiere_chaine, ...)
{
    va_list chaines;
    size_t longueur = strlen(premiere_chaine);
    char *resultat;
    char * pointeur_fin_resultat;

    va_start(chaines, premiere_chaine);
    for (;;)
    {
        char *p = va_arg(chaines, char *);

        if (p == NULL)
            break;
        longueur += strlen(p);
    }
    va_end(chaines);

    resultat = malloc(longueur + 1);
    EXIT_IF(resultat==NULL, ERR_MALLOC);

    strcpy(resultat, premiere_chaine);
    pointeur_fin_resultat = resultat + strlen(premiere_chaine);

    va_start(chaines, premiere_chaine);
    for (;;)
    {
        char *p = va_arg(chaines, char *);

        if (p == NULL)
            break;

        while (*p != '\0')
            *pointeur_fin_resultat++ = *p++;
    }
    va_end(chaines);
    *pointeur_fin_resultat = '\0';

    return resultat;
}
```

concatener.c

Comme d'habitude, le programme `concatener.h` contient le prototype de la fonction `concatener`. Il utilise la notation `...` présentée page 23.

concatener.h

```
#ifndef CONCATENER_H
#define CONCATENER_H

extern char *concatener(char *, ...);

#endif  /* CONCATENER_H */
```

concatener.h

Le programme suivant permet de tester la fonction `concatener` :

```
═══════ test_concatener.c ═══════
#include "concatener.h"

#define DEBUT "<<"
#define FIN   ">>"

main(int argc, char *argv[])
{
    char *s = concatener("Commande ", DEBUT, argv[0], FIN, NULL);

    puts(s);
    free(s);
}
═══════ test_concatener.c ═══════
```

Une exécution de cette commande produit le résultat suivant :

```
$ test_concatener
Commande <<test_concatener>>

$
```

Nous allons considérer un second exemple dans lequel le type des paramètres varie. Il s'agit d'une extension de la fonction précédente, en une fonction acceptant des paramètres de type caractère, entier, réel, ou pointeur de caractère, ce dernier cas codant les chaînes de caractères. Cette fonction ressemble à la fonction `sprintf`, mais alloue elle même la zone mémoire codant le résultat. À la différence de `sprintf`, elle n'impose pas de limitation sur la taille du résultat.

Nous allons en considérer une version simplifiée dans laquelle chaque paramètre est précédé d'une constante codant son type. Nous utiliserons quatre constantes de type : `Char`, `Int`, `Double`, `Str` et la constante de fin de liste `Eop`.[3] Pour illustrer son fonctionnement, considérons l'exemple suivant :

```
═══════ test_formater.c ═══════
#include <stdlib.h>

#include "hms.h"
#include "formater.h"

main(int argc, char **argv)
{
    int h;
    int m;
    int s;
    char *symbole;

    hms(&h, &m, &s);
    symbole = formater(Str, argv[0],
                       Str, "_",
                       Int,    h,
```

[3]Exceptionnellement, nous avons utilisé ici des identificateurs en anglais, du fait de leur association (du moins dans le cas des trois premiers) à des mots clés du langage.

```
                                Str, "_",
                                Int,    m,
                                Str, "_",
                                Int,    s,
                                Eop);

    printf("Symbole produit: \"%s\"\n", symbole);
    free(symbole);
}
```

test_formater.c

L'appel à la fonction `formater` qui y est effectué construit une chaîne de caractères en concaténant le nom de la commande avec les heures, minutes et secondes courantes. Voici un exemple de son exécution :

```
$ date ; test_formater
Thu Jun 17 00:32:45 MET DST 1993
Symbole produit: "test_formater_0_32_45"
$
$ mv test_formater commande
$ date ; commande
Thu Jun 17 00:33:02 MET DST 1993
Symbole produit: "commande_0_33_2"
$
```

Le paramètre fixe de la fonction est la première spécification de type. Toutes les spécifications sont définies dans le fichier `formater.h` au moyen de constantes énumérées.

formater.h

```
#ifndef FORMATER_H
#define FORMATER_H

enum format_specif
{
    Char, Int, Double, Str, Eop
};
typedef enum format_specif format_specif;

extern char *formater(format_specif, ...);

#endif  /* FORMATER_H */
```

formater.h

La lecture du source de la fonction `formater` ne devrait pas poser de problème particulier. Le mécanisme de récupération est similaire à celui de l'exemple précédent. La construction de la chaîne se fait par allocation/réallocation de mémoire, et le formatage utilise la fonction `sprintf`.

formater.c

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

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

#include "ERREURS.h"
#include "exit_if.h"

#include "formater.h"

#define TAILLE_MAX_INT    16
#define TAILLE_MAX_DOUBLE 32

#define TBLOC 256 - 16

static void agrandir();

static char   *format;
static int    position;
static int    taille;

char *formater(format_specif debut, ...)
{
    va_list        par;
    format_specif specification;
    char          *resultat;

    va_start(par, debut);
    specification = debut;

    format = malloc(TBLOC);
    EXIT_IF(format==NULL, ERR_MALLOC);
    position = 0;
    taille = TBLOC;

    while (specification != Eop)
    {
        switch (specification)
        {
          case Char:
            {
                char c = va_arg(par, char);

                agrandir(sizeof(char));
                sprintf(format+position, "%c", c);
                break;
            }

          case Int:
            {
                int i = va_arg(par, int);

                agrandir(TAILLE_MAX_INT);
                sprintf(format+position, "%d", i);
                break;
            }

          case Double:
            {
                double x = va_arg(par, double);
```

```
                agrandir(TAILLE_MAX_DOUBLE);
                sprintf(format+position, "%g", x);
                break;
            }

          case Str:
            {
                char *s = va_arg(par, char *);

                agrandir(strlen(s));
                sprintf(format+position, "%s", s);
                break;
            }
        }
        while (format[position] != '\0')
            position++;
        specification = va_arg(par, int);
    }
    resultat = malloc(strlen(format) + 1);
    strcpy(resultat, format);
    free(format);
    return(resultat);
}

static void agrandir(int extension)
{
    if (position + extension >= taille)
    {
        while (position + extension >= taille)
            taille += TBLOC;
        format = realloc(format, taille);
        EXIT_IF(format == NULL, ERR_REALLOC);
    }
}
```
formater.c

7.8 Lancement d'une commande

<stdlib.h>

Il est possible de transmettre une commande à l'interprète de commandes du système au moyen de la fonction

`int system(const char *`⟨*commande*⟩`)`

Si ⟨*commande*⟩ est la valeur `NULL`, la fonction retourne une valeur différente de zéro s'il existe dans cet environnement un mécanisme pour interpréter la commande. Lorsque ⟨*commande*⟩ est une chaîne de caractères, celle-ci est transmise à l'interprète. La valeur de retour est dépendante de l'implémentation.

L'exemple suivant est une fonction enchaînant le lancement de la commande **DATE(1)**, de l'interprète de commandes **SH(1)**, puis, une seconde fois, la commande **DATE(1)**.

```
═══════ exemp_system.c ═══════════════════════════
#include <stdlib.h>
#include <stdio.h>

main()
{
    printf("\n>>> Date de debut: ");
    fflush(stdout);
    system("date");
    printf("\n--- Entree dans un shell ---\n");
    system("sh");
    printf("--- Sortie du shell --------\n");
    printf("\n>>> Date de fin: ");
    fflush(stdout);
    system("date");
}
══════════════════════════════ exemp_system.c ═══════
```

La commande pt lancée dans le shell empilé est une version domaine public de la commande ps listant un ensemble de processus système en visualisant l'arborescence sous-jacente.

```
$ exemp_system
>>> Date de debut: Mon May 31 23:19:24 MET DST 1993
--- Entree dans un shell ---
$ pt p
 PID USER   STAT TT  TIME COMMAND
2694 achille IW  co  0:00 xinit
2695 achille S   co  9:42 | X :0
2697 achille IW  co  0:00 | | xconsole
2701 achille IW  co  0:01 | | xman
2702 achille IW  co  0:12 | | oclock
2711 achille S   co  1:12 | | xterm -ls -sb
2713 achille S   p1  0:19 | | | -bash (bash)
3503 achille IW  p1  1:47 | | | | emacs
3504 achille IW  p1  0:02 | | | | | /local/emacs/etc/wakeup 60
4139 achille IW  p1  0:32 | | | | xtex livre
4615 achille S   p1  0:00 | | | | exemp_system
4618 achille S   p1  0:00 | | | | | sh -c sh
4619 achille S   p1  0:00 | | | | | | sh
4623 achille R   p1  0:00 | | | | | | pt p
2726 achille S   co  0:29 | | twm
$ date
Mon May 31 23:19:34 MET DST 1993
$ [C-d] --- Sortie du shell --------

>>> Date de fin: Mon May 31 23:19:37 MET DST 1993
$
```

Le processus exemp-system a le numéro 4615; il a été lancé depuis le shell numéro 2713. Lorsque l'arborescence des processus est visualisée, le programme exemp-system vient d'exécuter l'instruction

```
system("sh");
```

qui lance un shell (le processus numéro **4618**) et lui fait exécuter la commande reçue en paramètre, c'est-à-dire **sh**. C'est donc sous cette nouvelle occurrence de l'interprète, le processus numéro **4619** qu'est effectuée la session située entre les lignes

```
--- Entree dans un shell ---
```

et

```
--- Sortie du shell --------
```

En particulier, c'est lui qui lance la commande `pt` visualisant l'arborescence.

Les programmes utilisant la fonction **system** *ne sont pas portables, la commande passée en paramètre étant dépendante du système. Pour faciliter le portage, on pourra définir la commande au moyen d'une pseudo-constante dans un fichier en-tête.*

7.9 Sauvegarde et restauration du contexte

Dans cette section, nous allons montrer comment on peut manipuler le contexte d'exécution d'un processus, ainsi que l'intérêt et les limitations de ce mécanisme.

7.9.1 Contexte d'exécution

Rappelons tout d'abord que les variables d'un programme peuvent être :

- des variables **permanentes** : ce sont les variables statiques, déclarées hors de tout bloc, ou dont la déclaration est explicitement préfixée par le mot-clé `static`;
- des variables **temporaires** : ce sont les variables dynamiques déclarées à l'intérieur d'un bloc et les paramètres de fonctions.

Sous UNIX, les variables permanentes sont regroupées dans une partition mémoire appelée la **région des données** (nous avons déjà évoqué la région du texte en 7.3). Elles ont la même durée de vie que le processus auquel elles sont attachées. C'est également dans la région des données que se trouvent les blocs alloués dynamiquement; cette région est extensible.

L'existence d'une variable temporaire est liée à l'exécution de la fonction qui la contient. Les variables temporaires sont allouées à l'appel de la fonction dans laquelle elles sont définies, et leur place est restituée lors du retour de cette fonction. À chaque appel d'une fonction, son **contexte** (c'est-à-dire ses paramètres, ses variables dynamiques et son adresse de retour) est empilé dans la **pile d'exécution** du programme. Si cette fonction appelle une nouvelle fonction, ou encore se rappelle récursivement elle-même, un nouveau contexte est empilé. Chaque retour de fonction entraine le dépilement du contexte courant, et le retour dans le contexte de la fonction appelante.

Sous UNIX, la pile d'exécution est implémentée dans une troisième zone mémoire appelée **région de pile**. Ces trois régions constituent le contexte utilisateur d'un processus.

Sur toute machine, certains registres sont réservés à la manipulation du contexte des processus. Les principaux sont

- le registre d'instruction qui pointe dans le code du programme vers la prochaine instruction à exécuter (PC),
- le pointeur de pile (SP),
- le mot d'état du processeur (PS),
- les registres contenant la valeur de retour d'une fonction.

Sauvegarder le contexte d'exécution d'un processus consiste tout simplement à sauvegarder la valeur des registres à un instant t donné. Il suffit de restaurer la valeur de ces registres pour remettre le processus dans l'état où il était à l'instant t.

7.9.2 Les fonctions setjmp et longjmp

`<setjmp.h>`

Les fonctions setjmp et longjmp permettent de manipuler le contexte d'exécution du programme, en réalisant des sauvegardes et des restaurations des registres machine. La restauration doit être effectuée dans la fonction F qui a également réalisé la sauvegarde, ou dans une fonction descendante de F dans l'arbre des appels. Dans ce cas :

- la restauration du registre de pile ramène le programme dans le contexte de F;
- la restauration de PC ramène, dans F, à l'endroit où a été effectuée la sauvegarde;
- enfin, la restauration du registre de retour de fonction à valeur entière détermine la valeur retournée par la fonction sur l'appel de laquelle on est artificiellement ramené.

La syntaxe d'appel de ces fonctions est :

```
int  setjmp(jmp_buf ⟨contexte⟩)
void longjmp(jmb_buf ⟨contexte⟩, int ⟨valeur-de-retour⟩)
```

Le paramètre ⟨*contexte*⟩ est un vecteur dans lequel sont sauvegardés les registres définissant le contexte du processus. Le type de ce paramètre, `jmp_buf`, est défini dans le fichier en-tête `<setjmp.h>`.

La sauvegarde de contexte est effectuée par la fonction `setjmp` qui retourne la valeur entière zéro. La restauration de ce contexte est effectuée en appelant la fonction `longjmp` avec la sauvegarde du contexte en paramètre; tout se passe alors comme si on était ramené au moment où a été effectué l'appel à la fonction `setjmp` : l'appel à la fonction `longjmp` est remplacé par un retour de la fonction `setjmp` à l'endroit ou a été effectuée la sauvegarde du contexte. La fonction `setjmp` retourne, cette fois, la valeur du second paramètre de `longjmp`, qui a été artificiellement recopiée dans la sauvegarde du registre de retour de fonction.

Considérons les deux instructions suivantes

```
(1) n = setjmp(cntx);
(2) longjmp(cntx, 1);
```

situées à différents endroits dans un programme. Lors de l'exécution de l'instruction (1), la fonction `setjmp` sauvegarde le contexte courant dans `cntx` et retourne la valeur zéro qui est alors affectée à la variable `n`. Ce contexte contient, entre autres, l'adresse de retour de l'appel à la fonction `setjmp`. Si plus tard dans le déroulement du programme l'instruction (2) est exécutée, le contexte sauvegardé dans `cntx` est restauré. La conséquence de cette restauration est la reprise du programme à l'adresse de retour de l'appel à `setjmp` en (1).

La seule différence entre le contexte sauvegardé et le contexte restauré est la valeur de retour de fonction. Lors du retour provoqué par (2), c'est le second paramètre de `longjmp`, c'est-à-dire la valeur 1, qui est retourné, et affecté à `n`. Il est donc possible de tester cette valeur pour déterminer si on est arrivé en (1) de *façon normale* ou à la suite d'une restauration de contexte.

L'exemple suivant montre une application typique de ce mécanisme. La fonction `appel_boucle` sauvegarde son contexte avant d'appeler la fonction `boucle` :

```
int appel_boucle()
{
    void boucle();
    int retour = setjmp(contexte);

    printf("setjmp retourne %d\n", retour);
    if (retour == 0)
        boucle();
    return retour;
}
```

Celle-ci exécute une boucle infinie, alternant la lecture d'une commande au moyen de la fonction `getcmd`, et son exécution – ici un simple affichage. La commande `q`, provoque la restauration du contexte.

```
void boucle()
{
    for (;;) {
        char getcmd(char *);
        char c = getcmd("-> ");

        switch (c) {
          case 'q':
            longjmp(contexte, (int) c);

          default:
            printf("Traitement de %c
            basln", c);
            break;
        }
    }
}
```

Le source de la fonction `getcmd` est élémentaire :

```
char getcmd(char *s)
{
    char c = (printf("%s", s), getchar());

    while (getchar() != '\n')
        ;
    return c;
}
```

L'exécution du programme se poursuit alors dans la fonction appel_boucle qui détecte la restauration de contexte et effectue un retour à la fonction appelante, dans ce cas la fonction main.

```
#include <setjmp.h>
jmp_buf contexte;

main()
{
    int appel_boucle();

    printf("Terminaison avec \"%c\"\n", appel_boucle());
}
```

Voici un exemple d'exécution du programme :

```
$ contexte
setjmp retourne 0
-> a
Traitement de a
-> q
setjmp retourne 113
Terminaison avec "q"
$
```

Pour mieux comprendre l'enchaînement des instructions de ce programme, nous allons suivre une partie de son exécution sous le débogueur symbolique, depuis l'entrée dans la fonction appel_boucle jusqu'à la sortie de la même fonction :

```
> appel_boucle()
14          int retour = setjmp(contexte);
16          printf("setjmp retourne %d\n", retour);
setjmp retourne 0
17          if (retour == 0)
18              boucle();
> boucle()
26              char c = getcmd("-> ");
-> a
28              switch (c) {
34                  printf("Traitement de %c\n", c);
Traitement de a
35                  break;
24          for (;;) {
26              char c = getcmd("-> ");
-> q
```

```
28              switch (c) {
31                  longjmp(contexte, (int) c);
>> appel_boucle()
16          printf("setjmp retourne %d\n", retour);
setjmp retourne 113
17          if (retour == 0)
19          return (char) retour;
20      }
< appel_boucle()
```

Les lignes en italique indiquent les entrées et les sorties de fonction. On n'a pas fait figurer le déroulement de la fonction `getcmd`.

7.9.3 Limites du mécanisme de sauvegarde/restauration

La restauration de contexte ne peut se faire qu'en dépilant un empilement d'appels de fonction. Il n'est possible de restaurer un contexte que s'il a été sauvegardé dans une fonction présente dans la pile des appels, c'est-à-dire la fonction courante, ou la fonction appelante de la fonction courante, ou la fonction appelante de la fonction appelante de la fonction courante...

Considérons l'exemple suivant :

1) la fonction `main` appelle une fonction `f1`;
2) la fonction `f1` appelle une fonction `f2`;
3) la fonction `f2` sauvegarde son contexte et exécute un retour de fonction;
4) la fonction `f1` tente de restaurer le contexte sauvegardé par `f2`.

Cette restauration est incorrecte; elle est incompatible avec la pile des appels de fonction. La figure 7.3 visualise la pile au moment de la sauvegarde et au moment de la restauration. La session suivante montre le source d'un tel programme et son exécution.

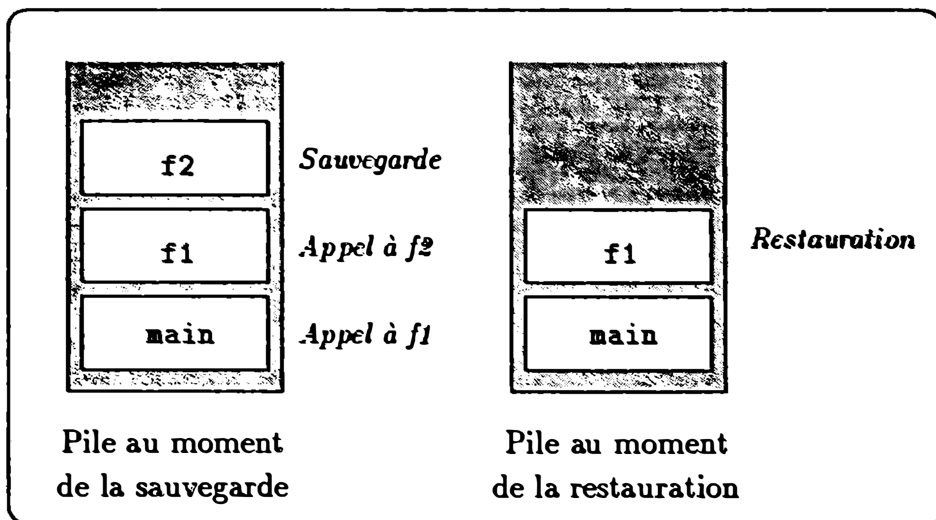


Figure 7.3: Sauvegarde et restauration de la pile d'exécution

err_contexte.c

```
#include <stdio.h>

#include <setjmp.h>
jmp_buf contexte;

void f1();
void f2();

main()
{
    printf("Appel a f1\n");
    f1();
    printf("Retour de f1\n");
}

void f1()
{
    printf("Appel a f2\n");
    f2();
    longjmp(contexte, 1);
}

void f2()
{
    if (setjmp() == 0)
        return;
    printf("Retour dans f2\n");
}
```

err_contexte.c

```
$ err_contexte
Appel a f1
Appel a f2
Segmentation fault (core dumped)
$
```

Une autre utilisation illégale est la suivante :

```
main:
appel f1
    f1:
    sauvegarde du contexte
    retour
appel f2
    f2:
    restauration du contexte
```

En résumé, ces fonctions permettent seulement de dépiler une suite d'appels de fonctions. Elles sont particulièrement adaptées au traitement d'erreur, et à la gestion de points de reprise dans un traitement complexe.

Chapitre 8

Modularisation des programmes C

La **modularisation** d'un programme a pour but son découpage en composants indépendants, et si possible de petite taille. Si tout le monde s'accorde à dire que cette approche est indispensable pour produire du logiciel de qualité, sa mise en œuvre soulève des problèmes variés et fait encore l'objet de controverses. Différents styles, non nécessairement compatibles, ont vu le jour, depuis l'approche fonctionnelle descendante, héritée de la programmation dite *structurée*, et qui ne présente plus guère qu'un intérêt historique.

Dans ce chapitre, nous allons passer en revue plusieurs approches permettant de modulariser les applications en langage C. Nous nous plaçons donc ici du point de vue de la *méthodologie de programmation*, et non de celui des *méthodologies d'analyse*. Par conséquent, nous n'abordons pas le problème de la modularisation du point de vue de la conception des logiciels, mais plutôt de celui de la qualité du code, dans un sens que nous allons préciser.

8.1 Quelques principes de modularité

8.1.1 Définition des objectifs

Commençons par définir les objectifs que nous souhaitons atteindre. Nous en retiendrons cinq.

1) La **réutilisabilité** : la possibilité d'assembler entre eux plusieurs portions de programme, écrites à des époques et pour des applications différentes, sans qu'aucune d'entre elles ne perturbe le comportement des autres.
2) L'**extensibilité** : la possibilité de modifier l'implémentation d'une partie du programme, ou de lui ajouter des fonctionnalités, sans que cela ne modifie le comportement de l'ensemble.
3) La **portabilité** : la facilité à adapter toutes les fonctionnalités d'un logiciel à un nouvel environnement.
4) La **maintenabilité** : la facilité à corriger des erreurs dans une portion de programme sans en introduire de nouvelles dans le reste de l'application.
5) La **lisibilité** : la facilité à appréhender le comportement d'un programme et à comprendre sa mise en œuvre.

On considérera qu'un programme est **modulaire** lorsqu'il est éclaté en composants, les **modules**, satisfaisant ces cinq objectifs. L'idée de base est de regrouper ensemble les fonctions associées à un même traitement. La manière de former ces regroupements s'apparente aux méthodes d'analyse, ce qui n'est pas, comme nous l'avons déjà dit, l'objet de ce chapitre. Signalons simplement deux approches qui s'opposent :

- l'approche *dirigée par le contrôle*, consistant à dégager les traitements, en recherchant la fonction principale du logiciel que l'on subdivise ensuite en sous-fonctions, *sous*-sous-fonctions...
- l'approche *dirigée par les données*, consistant à dégager les données et la façon dont elles interfèrent entre elles.

La première approche aboutit à l'*analyse fonctionnelle descendante*, incompatible avec les objectifs de réutilisabilité et d'extensibilité. La seconde approche est une *approche ascendante*, favorisant la construction de composants logiciels autonomes, et que l'on peut assembler afin de former des composants d'un niveau d'abstraction de plus en plus élevé. Les méthodes d'*analyse par objets*, et les techniques de *programmation par objets* (ou P.P.O.) associées sont l'illustration de cette démarche.

Nous allons dans la suite de ce chapitre présenter différentes manières de réaliser en C l'implémentation de modules, c'est-à-dire de composants de programme satisfaisant les objectifs énoncés plus haut. Ces méthodes de codage seront illustrées à l'aide d'exemples de modules relativement simples et/ou classiques. Elles sont bien adaptées à l'implémentation de modules définis par des méthodes d'analyse dirigée par les données, et en particulier les *approches objets*.

8.1.2 Principes de base

On peut mettre en évidence trois principes fondamentaux d'**abstraction modulaire** qui guideront toute cette approche :

1) **l'abstraction de constantes littérales**
2) la **factorisation de code**
3) le **masquage d'implémentation**

Abstraction de constantes littérales

L'utilisation explicite de constantes littérales (3.1.2) dans le code d'une fonction constitue une entrave à l'extensibilité et à la maintenabilité du logiciel. Par conséquent, des constructions comme

```
annee_complete = concatener("19", annee);
write(fd, "-> ", 3);
fopen("/users/bidru/lib/mag.scores", "r");
```

doivent être considérées comme des fautes de programmation. Dans la première instruction, la chaîne `"19"` devrait être définie par une constante, ou par une variable initialisée par défaut avec une constante. La seconde est doublement à rejeter, d'une part pour la présence de la chaîne de caractères `"->"`, et d'autre part pour la constante `3` qui devrait être calculée automatiquement à partir de la précédente. La dernière des trois instructions ne permet pas le portage du

logiciel dans un environnement différent de celui dans lequel il a été développé, sans modification de code. Il y a plusieurs façons d'y remédier :

- placer la définition du chemin dans un fichier en-tête;
- associer le chemin à une pseudo-constante définie à la compilation avec l'option `-D` (voir page 153); dans ce cas, la valeur de la constante pourra être définie dans le fichier `Makefile`;
- initialiser le chemin avec une variable de l'environnement utilisateur (voir 7.5);

Il y a plusieurs cas particuliers de constantes littérales qu'il est naturel d'utiliser explicitement dans le code, comme certaines constantes dont la sémantique est fixée : `\n`, `\0`, `0`, `1`... C'est également le cas des spécifications de format passées en paramètres aux fonctions `printf` et `scanf`, qui sont des constantes littérales chaînes de caractères. Il faut par contre prendre garde, comme nous allons le voir dans le paragraphe suivant, à éviter les duplications de ces spécifications.

⊽ *Sauf cas très particuliers, les constantes littérales doivent être abstraites au moyen de constantes symboliques ou de variables initialisées avec une constante.*

Factorisation de code

Le but de la **factorisation** est d'éviter les **duplications de code**, c'est-à-dire la présence d'une construction autre qu'un symbole en plusieurs endroits du programme. Toute duplication de code est un obstacle à la maintenabilité et l'extensibilité d'un programme. Ce principe est tellement évident qu'il est semble ressortir du lieu commun. Malheureusement, il n'est encore, dans la pratique informatique quotidienne, que très peu suivi.

Il peut s'agir de la duplication de constantes littérales, comme dans l'exemple suivant :

```
int pile[100];
    :
if (indice_de_pile < 100)
{
    ...
```

où la constante entière `100` est répétée deux fois. Dans ce cas, l'utilisation d'une pseudo-constante

```
#define TAILLE_PILE 100
```

ou

```
const int taille_pille = 100;
```

s'impose. De toutes façons, si l'on respecte le principe précédent d'abstraction des constantes littérales, le problème ne se pose plus.

On peut rencontrer des exemples plus délicats, dans lesquels la duplication est moins évidente. Considérons un logiciel interactif effectuant la lecture d'un

caractère de commande et utilisant un aiguillage pour sélectionner le traitement :

```
c = caractere_suivant();
switch (c)
{
  case 'q':
    ⋮
```

Ce logiciel offre également une fonctionnalité d'aide, avec l'impression de messages de la forme

```
printf("q : sauvegarde et sortie\n");
```

La constante caractère `'q'` apparaît par conséquent deux fois, une première fois dans le sélecteur de l'aiguillage, et une seconde dans la chaîne du message d'aide. Cette duplication peut être résolue de la façon suivante

```
#define QUIT 'q'

case QUIT:

printf("%c : sauvegarde et sortie\n", QUIT);
```

Il est également préférable d'effectuer l'abstraction du message de sortie par

```
#define QUIT_MESSAGE "sauvegarde et sortie"

printf("%c : " QUIT_MESSAGE "\n", QUIT);
```

ou encore

```
char *quit_message = ...

printf("%c : %s\n", QUIT, quit_message);
```

Nous en verrons un exemple plus complet page 276.

Une autre faute de duplication, que l'on rencontre très fréquemment, concerne les chaînes de caractères de formats. Dans ce cas, la manière la plus simple d'y remédier est de factoriser le code au moyen d'une fonction. Par exemple dans le programme

```
printf("%g+%gi", x.a, x.b);
          ⋮
printf("%g+%gi",
       partie_reelle(ro,teta),
       partie_imaginaire(ro,teta));
          ⋮
```

les appels à `printf` doivent être remplacés par des appels à une fonction d'impression :

```
complexe_imprimer(x.a, x.b)
```

et

```
complexe_imprimer(partie_reelle(ro,teta),
                  partie_imaginaire(ro,teta));
```

définie par

```
void complexe_imprimer(double a, double b)
{
    printf("%g+%gi", a, b);
}
```

Les fonctions doivent être systématiquement employées pour éviter la duplication de code, même dans le cas où la duplication se limite à une seule instruction. Si cela s'avère nécessaire, il ne faut pas craindre de définir une multitude de très petites fonctions.

En conclusion, il est fondamental lors de toute la phase de codage d'un programme[1] de traquer toutes les duplications de code, sous quelque forme que ce soit, et de les éliminer en utilisant les abstractions modulaires que sont les constantes et les fonctions.

Masquage de l'implémentation

Le troisième principe fondamental de la modularité est celui du **masquage de l'implémentation**. Il consiste à séparer clairement, pour chaque module, la partie qui réalise son interface de celle qui réalise son implémentation, et à exporter seulement la première hors du module. On peut visualiser cette organisation au moyen du schéma de la figure 8.1. Le module A utilise les modules B et C, et le module C utilise le module B. Chaque module a seulement accès à l'interface des deux autres, c'est-à-dire à la manière de mettre en œuvre ses fonctionnalités. De cette façon, une modification de l'implémentation d'un module qui ne change pas son comportement n'a pas d'incidence sur le reste du programme.

Cette organisation favorise de façon évidente la réutilisabilité et la maintenabilité. De même, il devient possible d'accroître les fonctionnalités d'un module de manière à ce que l'ancienne interface soit incluse dans la nouvelle, ce qui favorise l'extensibilité. On parle alors de **compatibilité ascendante**.

Un découpage modulaire implémentant ce principe nécessite un mécanisme de définition de variables et de fonctions locales. De manière générale, les structures de données du module sont des informations privées de ce module, et font partie de son implémentation : on parle dans ce cas d'**encapsulation des données**. L'interface sera implémentée au moyen de fonctions qui auront seules le droit d'agir sur les données du module.

En C, l'encapsulation de données peut être effectuée de deux façons différentes, soit au moyen d'une fonction, soit au moyen d'un fichier. Dans la suite de ce chapitre, nous allons considérer les deux cas de figure.

8.2 La fonction comme unité modulaire

8.2.1 Fonctions pures

Il s'agit du cas le plus simple, celui de fonctions C implémentant des fonctions au sens mathématique du terme. Deux appels consécutifs, effectués avec les

[1] En langage C ou dans tout autre langage.

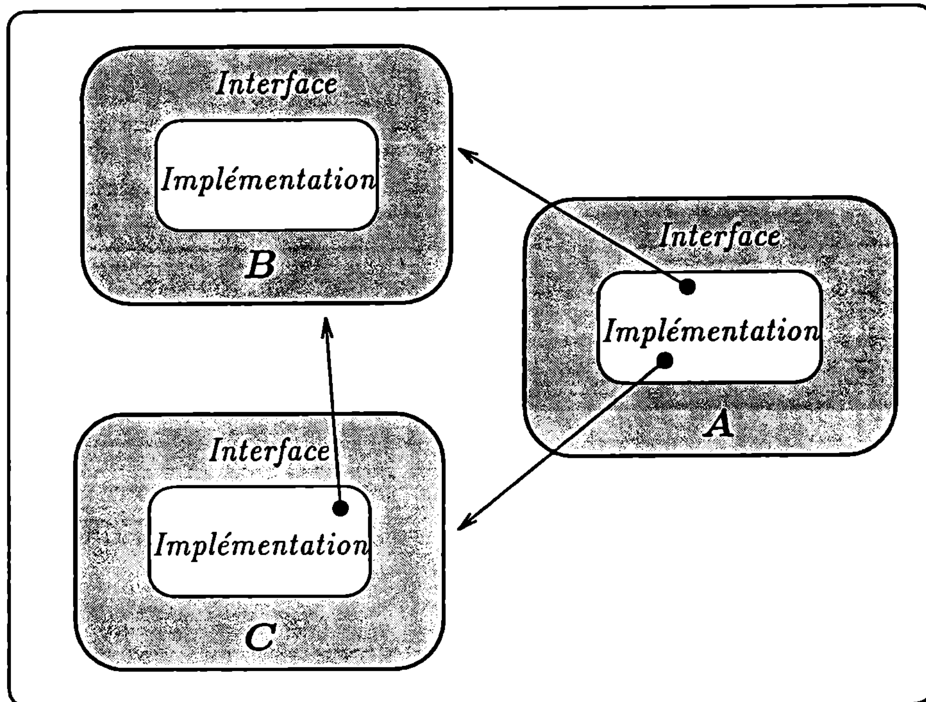


Figure 8.1: Interaction entre plusieurs modules

mêmes paramètres, produisent le même résultat. Dans ce cas, chaque fonction constitue un module élémentaire indépendant et autonome. Le regroupement de plusieurs fonctions dans un module plus gros est purement arbitraire. Par exemple, on pourra définir un module `arithmetique` contenant les fonctions `pgcd`, `ppcm`, etc. Un tel module est simplement une bibliothèque de fonctions.

Remarque 29 *Pour être tout à fait exact, une fonction n'est pure que si elle n'effectue aucun effet de bord sur l'environnement externe à cette fonction, en particulier sur des variables de l'environnement global ou sur des paramètres passés par référence. On dit dans ce cas qu'elle respecte le principe de* **transparence référentielle**.

8.2.2 Fonctions avec environnement

Il est possible d'associer un environnement permanent à une fonction, c'est-à-dire un ensemble de variables dont la durée de vie est celle du programme. Pour cela, on donne à ces variables un statut de variables permanentes à l'aide du mot-clé `static` (voir 2.6.2).

Nous allons considérer, pour commencer, l'exemple très simple d'un module produisant, chaque fois qu'il est invoqué, un nouveau nombre entier. De tels objets sont généralement appelés des **générateurs**. Il s'agit donc ici d'un *générateur d'entiers*.

Définition de l'interface

L'interface du module `generateur` est composée de l'unique fonction implémentant ce module; elle est décrite dans le fichier en-tête `generateur.h`.

generateur.h *(élémentaire)*
```
#ifndef GENERATEUR_H
#define GENERATEUR_H

extern int generateur(void);

#endif  /* GENERATEUR_H */
```
generateur.h *(élémentaire)*

Définition de l'implémentation

L'implémentation du module generateur, c'est-à-dire la définition de la fonction de même nom, est contenue dans le fichier source generateur.c.

generateur.c *(élémentaire)*
```
#include "generateur.h"

#define VALEUR_INITIALE 1

int generateur()
{
    static int valeur = VALEUR_INITIALE;

    return valeur++;
}
```
generateur.c *(élémentaire)*

La variable valeur est locale à la fonction generateur, et permanente. Elle est initialisée lors de la compilation du programme (voir 2.7.1). Le premier appel à la fonction generateur retourne l'entier 1, et chaque nouvel appel retourne un entier, consécutif du précédent. La valeur d'initialisation est définie au moyen d'une pseudo-constante.

Remarque 30 *L'inclusion, dans le fichier* generateur.c*, du fichier en-tête* generateur.h *peut sembler redondante. En réalité, elle est fondamentale, car elle permet au compilateur de vérifier que la description de l'interface, c'est-à-dire le prototype de ses fonctions, est bien conforme à son implémentation.*

Exemple d'utilisation

Un avantage de cette forme est de ne nécessiter aucune initialisation. Voici l'exemple d'un second module, le module gen_ ymb générant à chaque appel un symbole de la forme

__GLB_*i*__

Il utilise la fonction formater définie en 7.7, page 243. Comme précédemment, voici l'interface du module :

gen_symb.h
```
#ifndef GEN_SYMB_H
#define GEN_SYMB_H
```

```
extern char *gen_symb(void);

#endif  /* GEN_SYMB_H */
```
gen_symb.h

et son implémentation :

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

#include "generateur.h"
#include "formater.h"

#define PREFIXE_SYMBOLE "__GLB_"
#define SUFFIXE_SYMBOLE "__"

char *gen_symb()
{
    int n = generateur();
    char *symbole = formater(Str, PREFIXE_SYMBOLE,
                             Int, n,
                             Str, SUFFIXE_SYMBOLE,
                             Eop);

    return(symbole);
}
```
gen_symb.c

Le programme suivant teste ce module. On vérifiera que le principe de non-duplication de code est respecté.

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

#include "pluriel.h"
#include "gen_symb.h"

static void traitement(void);

main(int argc, char *argv[])
{
    int n = argc > 1 ? atoi(argv[1]) : 0;
    char *symb;

    printf("Generation d'un premier symbole\n");
    traitement();

    printf("Generation de %d symbole%s\n", n, PLURIEL(n));
    while (n--)
        traitement();

    printf("Generation d'un dernier symbole\n");
    traitement();
}
```

```
static void traitement(void)
{
    char *symb = gen_symb();

    printf("  \"%s\"\n", symb);
    free(symb);
}
```
symboles.c

Une exécution de ce programme produit le résultat suivant :

```
$ symboles 4
Generation d'un premier symbole
  "__GLB_1__"
Generation de 4 symboles
  "__GLB_2__"
  "__GLB_3__"
  "__GLB_4__"
  "__GLB_5__"
Generation d'un dernier symbole
  "__GLB_6__"
$
```

Il est clair que l'on respecte ici pleinement le principe de masquage de l'implémentation. Une modification de l'implémentation de n'importe lequel des modules ne changeant pas son comportement ne change pas, non plus, celui du reste du programme. Par exemple, on peut récrire `generateur` de la façon suivante :

`generateur.c` *(élémentaire)*

```
#include "generateur.h"

#define VALEUR_INITIALE 0

int generateur()
{
    static int valeur = VALEUR_INITIALE;

    valeur = valeur+1;
    return valeur;
}
```
`generateur.c` *(élémentaire)*

Cette nouvelle version du module `generateur` peut-être utilisé sans modifier le comportement du module `gen_symb`, ce qui n'aurait pas été le cas si ce dernier avait accédé directement à la variable `valeur` de `generateur`.

On peut appliquer ce principe pour modulariser certains des programmes présentés dans le début de cet ouvrage. Par exemple, à partir du programme `majuscule` présenté page 212, nous allons écrire la fonction `typo_phrase` qui reçoit un caractère `c` en paramètre, et qui

- lors de son premier appel, retourne la majuscule de `c` si `c` est une minuscule, et `c` sinon;

– lors d'un autre appel, retourne la majuscule ou la minuscule de `c` selon que l'on se trouve ou non en début d'une phrase.

Voici l'implémentation de ce module :

```
══════════ typo.c ══════════════════════════════
#include <ctype.h>
#include "typo.h"
#define IS_END_OF_SENTENCE(c)  ((c)=='.' || (c)=='?' || (c)=='!')

char typo_phrase(char c)
{
    static int debut_phrase = 1;

    if (debut_phrase)
    {
        debut_phrase = !isalpha(c);
        if (islower(c))
            return toupper(c);
        return c;
    }
    if (isupper(c))
        return tolower(c);
    debut_phrase = IS_END_OF_SENTENCE(c);
    return c;
}
══════════════════════════════════ typo.c ══════
```

Cette transformation de la commande `majuscule` avec la définition d'un module effectuant le traitement typographique favorise la lisibilité du code et sa réutilisabilité. Il est très facile de récrire la commande de la façon suivante :

```
═════════ majuscule.c ══════════════════════════
#include <stdio.h>

#include "typo.h"

main()
{
    for (;;)
    {
        char c = getchar();

        if (c == EOF)
            break;
        putchar(typo_phrase(c));
    }
}
═════════════════════════════ majuscule.c ══════
```

Ce nouveau programme a gagné en lisibilité grâce à la séparation effectuée entre les opérations d'entrées-sorties et le traitement des données. Le module `typo` peut d'autre part être réutilisé, par exemple pour traiter une chaîne de caractères en mémoire :

```
═══════ test_typo.c ═══════════════════════════
#include <stdio.h>
#include <stddef.h>

#include "typo.h"
#include "lire_chaine.h"

main()
{
    char *s = lire_chaine("Entrer une phrase: ");
    char *p;

    if (s == NULL)
        exit(0);

    for (p=s; *p; p++)
        *p = typo_phrase(*p);

    printf(" --> \"%s\"\n", s);
}
═══════════════════════════════ test_typo.c ═══════
```

Ce programme utilise la fonction `lire_chaine` définie en 7.3.

```
$ typo
typo
Entrer une phrase: ou il est le chat ?  ah !  \
il est la... [tristan, acte 2]
 --> "Ou il est le chat ?  Ah !  Il est la... [Tristan, acte 2]"
$
```

8.2.3 Gestion de plusieurs fonctionnalités

Envoi de message

La limite évidente de l'implémentation que nous venons de présenter est de coder une seule fonctionnalité par module. Si l'on reprend le cas du générateur d'entiers, il est assez naturel de souhaiter pouvoir le réinitialiser afin de produire une nouvelle suite. Pour cela, nous allons paramétrer chaque appel au générateur par un message, implémenté sous la forme d'une constante symbolique, et permettant de sélectionner l'une des actions parmi celles implémentées dans le module.

Dans le cas de notre générateur d'entiers, nous allons considérer deux messages :

- `Aller_au_debut`, qui réinitialise le générateur;
- `Suivant`, qui produit un nouvel élément de la suite.

Voici la nouvelle interface du module `generateur` :

```
═══════ generateur.h (avec messages) ═══════════════
#ifndef GENERATEUR_H
#define GENERATEUR_H
```

```
enum message
{
    Aller_au_debut, Suivant
};

typedef enum message message;

extern int generateur(message);

#endif  /* GENERATEUR_H */
```

generateur.h *(avec messages)*

L'implémentation est effectuée au moyen d'un aiguillage, jouant le rôle de sélecteur de message.

generateur.c *(avec messages)*

```
#include "generateur.h"

#define VALEUR_INITIALE    1

int generateur(message m)
{
    static int valeur = VALEUR_INITIALE;

    switch (m)
    {
      case Aller_au_debut:
        valeur = VALEUR_INITIALE;
        return;
        /* NOBREAK */

      case Suivant:
        return valeur++;
        /* NOBREAK */
    }
}
```

generateur.c *(avec messages)*

Le fichier suivant contient l'implémentation de la fonction `e_suite`, qui produit une suite d'entiers dont elle reçoit la longueur en paramètre. Cette fonction commence par initialiser le générateur avec le message `Aller_au_debut`, puis itère l'envoi du message `Suivant`.

e_suite.c

```
#include "e_suite.h"
#include "generateur.h"

void e_suite(int longueur)
{
    generateur(Aller_au_debut);
    while (longueur-- > 0)
        printf("%d ", generateur(Suivant));
}
```

e_suite.c

Le résultat suivant :

```
$ triangle_croissant 7
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4 5 6
1 2 3 4 5 6 7
$
```

est produit par le programme

```
======= triangle_croissant.c ==========================
#include <stdlib.h>

#include "e_suite.h"

main(int argc, char *argv[])
{
    int n = argc > 1 ? atoi(argv[1]) : 1;
    int longueur;

    for (longueur = 1; longueur <= n ; longueur++)
    {
        e_suite(longueur);
        putchar('\n');
    }
}
========================== triangle_croissant.c =======
```

Messages paramétrés

Pour être pleinement satisfaisant, ce mécanisme doit pouvoir être paramétré. On va pour cela ajouter la possibilité d'associer des paramètres aux différents messages. Nous allons étendre le comportement du générateur d'entiers de la façon suivante :

- `Aller_au_debut` : inchangé;
- `Suivant` : inchangé;
- `Definir_premier` ⟨*entier*⟩ : définition du premier élément de la suite;
- `Definir_pas` ⟨*entier*⟩ : définition de la raison de la suite.

La mise en œuvre utilise le mécanisme de fonctions avec paramètres variables (voir 7.7). L'interface du module `generateur` devient :

```
======= generateur.h (avec paramètres) ==================
#ifndef GENERATEUR_H
#define GENERATEUR_H

enum message
{
```

```
    Aller_au_debut, Suivant, Definir_premier, Definir_pas
};

typedef enum message message;

extern int generateur(message, ...);

#endif  /* GENERATEUR_H */
```
generateur.h *(avec paramètres)*

L'environnement de la fonction se compose maintenant de trois variables codant respectivement la valeur initiale de la suite, la prochaine valeur à produire, et le pas d'incrémentation. Le traitement des messages `Definir_premier` et `Definir_pas` est similaire :

- initialisation du parcours des paramètres de la fonction à partir du paramètre suivant le message (`va_start`),
- récupération d'un paramètre de type entier (`va_arg`),
- terminaison du parcours (`va_end`).

generateur.c *(avec paramètres)*
```
#include <stdarg.h>

#include "generateur.h"

#define VALEUR_INITIALE    1
#define PAS_PAR_DEFAUT     1

int generateur(message m, ...)
{
    static int valeur_initiale = VALEUR_INITIALE;
    static int valeur_courante = VALEUR_INITIALE;
    static int pas = PAS_PAR_DEFAUT;
    va_list parametres;

    switch (m)
    {
      case Definir_premier:
        va_start(parametres, m);
        valeur_initiale = va_arg(parametres, int);
        va_end(parametres);
        break;

      case Definir_pas:
        va_start(parametres, m);
        pas = va_arg(parametres, int);
        va_end(parametres);
        break;

      case Aller_au_debut:
        valeur_courante = valeur_initiale;
        break;

      case Suivant:
        {
            int n = valeur_courante;
```

```
            valeur_courante += pas;
            return n;
        }
        /* NOBREAK */
    }
}
```

generateur.c *(avec paramètres)*

Remarque 31 *On se trouve ici dans un cas où il est difficile de factoriser les trois instructions de récupération des paramètres, à cause de l'implémentation de la pseudo-fonction* `va_start`*. On pourrait factoriser à l'aide d'une nouvelle pseudo-fonction.*

On notera qu'il s'agit ici d'une extension d'interface avec *compatibilité ascendante* : toute application compilée avec la version 1 du module `generateur` (celle avec messages sans paramètre) peut être recompilée avec la version 2 (avec paramètres), sans que cela n'altère son comportement. En particulier, la fonction `e_suite` de l'exemple précédent peut être réutilisée en l'état. Par contre, il est maintenant possible d'utiliser les nouvelles fonctionnalités du module `generateur`, en définissant une nouvelle suite avant d'appeler la fonction `e_suite`. Le résultat suivant

```
$ triangle_decroissant 7
7
7 6
7 6 5
7 6 5 4
7 6 5 4 3
7 6 5 4 3 2
7 6 5 4 3 2 1
$
```

est produit par le programme

triangle_decroissant.c

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

#include "e_suite.h"
#include "generateur.h"

static void genere(int, int);

main(int argc, char *argv[])
{
    int n = argc > 1 ? atoi(argv[1]) : 1;
    int longueur;

    generateur(Definir_pas, -1);
    generateur(Definir_premier, n);
```

```
    for (longueur = 1; longueur <= n ; longueur++)
    {
        e_suite(longueur);
        putchar('\n');
    }
}
```

triangle_decroissant.c

8.2.4 Limites de l'implémentation de modules par des fonctions

Bien que la plupart des objectifs énoncés plus haut soient atteints, ce modèle n'est pas pleinement satisfaisant. Une première raison concerne l'organisation du code, à savoir une fonction renfermant un aiguillage dont la taille augmente à chaque extension du module. Cela nuit à la lisibilité du programme, et ne permet pas d'effectuer un découpage modulaire satisfaisant du module lui-même. Une amélioration consisterait à associer une fonction locale à chaque traitement, l'aiguillage se limitant dans chaque cas à un appel à la fonction concernée. Cette solution est assez lourde car elle impose de passer les variables de l'environnement en paramètre à chaque fonction locale. De plus, le passage doit s'effectuer par référence pour que les fonctions appelées puissent effectuer des effets de bord sur cet environnement, ce qui alourdit encore l'écriture.

Il existe une autre raison en défaveur de cette approche : le type de la fonction implémentant le module varie avec le message. Sur l'exemple précédent, elle est tantôt de type `int`, et tantôt de type `void`. Ce serait beaucoup plus gênant si elle prenait plusieurs types définis, `int` et `float` par exemple, car il serait nécessaire d'utiliser un mécanisme de type variable comme celui présenté en 2.4.3.

En conclusion, on atteint très rapidement les limites de cette approche qui peut être intéressante dans des cas particuliers, mais ne peut être adoptée comme un principe général d'implémentation de modules en C. Nous allons maintenant explorer la seconde possibilité utilisant le fichier comme unité modulaire.

8.3 Le fichier comme unité modulaire

8.3.1 Mise en œuvre

Le défaut principal du précédent modèle réside dans son incapacité à décomposer l'interface en un ensemble de petites fonctions indépendantes. Sur l'exemple précédent du générateur d'entiers, ce découpage aboutirait à quatre fonctions :

```
definir_premier
definir_pas
aller_au_debut
suivant
```

Ces fonctions doivent partager le même environnement permanent, à savoir les variables

```
valeur_initiale
valeur_courante
pas
```

La déclaration de ces variables doit par conséquent être effectuée au niveau zéro (voir 2.6.1), afin qu'elles soient visibles à l'intérieur de chacune des quatre fonctions. Ces variables doivent être locales au module (principe du masquage de l'implémentation). Nous savons (voir page 75) qu'il suffit pour cela d'utiliser le mot clé `static`. Seules les fonctions de l'interface sont globales.

Dans le modèle précédent, reposant sur une seule fonction d'interface, le nom de cette fonction est celui du module. Il est raisonnable d'imposer que chaque module possède un nom unique, afin de prévenir tout problème de collision de nom de fonction[2]. Avec ce nouveau modèle, le problème de la collision se pose à nouveau. Plus précisément, deux modules distincts peuvent posséder des fonctions d'interface ayant un même nom. Sur notre exemple, le module `generateur` possède parmi ses fonctions d'interface la fonction `aller_au_debut`. Il est fort probable que d'autres modules possèdent une fonction de même nom : `liste`, `buffer`, `fenetre_texte`....

Il est donc **impératif** d'adopter une convention permettant d'associer de manière unique chaque fonction d'interface à son module. La plus naturelle, qui est aussi la plus courante consiste à préfixer le nom de la fonction avec celui du module. Dans le cas précédent, cela donne :

═══ `generateur.h` *(module fichier)* ═══
```
#ifndef GENERATEUR_H
#define GENERATEUR_H

extern void generateur_definir_premier(int);
extern void generateur_definir_pas(int);
extern void generateur_aller_au_debut(void);
extern int  generateur_suivant(void);


#endif  /* GENERATEUR_H */
```
═══ `generateur.h` *(module fichier)* ═══

Comme précédemment, l'interface du module `generateur` est placée dans le fichier en-tête `generateur.h`, et son implémentation dans le fichier source `generateur.c`. Cette dernière est calquée sur la version précédente, avec un gain manifeste en modularité lié au découpage des traitements en fonctions indépendantes.

═══ `generateur.c` *(module fichier)* ═══
```
#include "generateur.h"

#define VALEUR_INITIALE    1
#define PAS_PAR_DEFAUT     1

static int valeur_initiale = VALEUR_INITIALE;
```

[2]Si deux modules possèdent un même nom, et que ce nom ne peut être modifié, il est vraisemblable qu'ils implémentent tous deux une même spécification (pile, liste...). Il est dans ce cas raisonnable d'exclure leur utilisation simultanée.

```
static int valeur_courante = VALEUR_INITIALE;
static int pas             = PAS_PAR_DEFAUT;

void generateur_definir_premier(int v)
{
    valeur_initiale = v;
}

void generateur_definir_pas(int p)
{
    pas = p;
}

void generateur_aller_au_debut()
{
    valeur_courante = valeur_initiale;
}

int generateur_suivant()
{
    int n = valeur_courante;

    valeur_courante += pas;
    return n;
}
```

generateur.c *(module fichier)*

Voici un exemple de commande utilisant ce module

autre_triangle.c

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

#include "generateur.h"
#include "e_suite.h"

main(int argc, char *argv[])
{
    int max = argc > 1 ? atoi(argv[1]) : 1;
    int i;

    for (i = 1; i <= max; i++)
    {
        generateur_definir_premier(1);
        generateur_definir_pas(1);
        e_suite(i);

        generateur_definir_premier(i-1);
        generateur_definir_pas(-1);
        e_suite(i-1);

        putchar('\n');
    }
}
```

autre_triangle.c

et le résultat de son exécution

```
$ autre_triangle 7
1
1 2 1
1 2 3 2 1
1 2 3 4 3 2 1
1 2 3 4 5 4 3 2 1
1 2 3 4 5 6 5 4 3 2 1
1 2 3 4 5 6 7 6 5 4 3 2 1
$
```

Nous sommes maintenant en mesure de proposer une méthode d'implémentation d'un module par un fichier :

(1) Un module ⟨*mod*⟩ est composé d'un fichier ⟨*mod*⟩`.h` définissant son interface, et d'un module ⟨*mod*⟩`.c` définissant son implémentation.
(2) L'interface est constituée par un ensemble de fonctions dont le nom est de la forme ⟨*mod*⟩`_...` appelées les *fonctions d'interface.*
(3) Le fichier ⟨*mod*⟩.h se compose
   - des prototypes des fonctions d'interface du module ⟨*mod*⟩;
   - d'éventuelles définitions de symboles (pseudo-constantes ou constantes énumérées);
   - d'éventuelles directives : inclusions de fichiers, pseudo-fonctions et directives de compilation conditionnelle.
(4) L'implémentation du module se compose
   - de variables de niveau zéro locales au fichier ⟨*mod*⟩`.c` et constituant son environnement permanent;
   - des fonctions dont le prototype se trouve dans le fichier ⟨*mod*⟩`.h`;
   - éventuellement de fonctions auxiliaires, locales au fichier ⟨*mod*⟩`.c`.
(5) Le fichier ⟨*mod*⟩`.h` est inclus dans le fichier ⟨*mod*⟩`.c`, et dans tout autre module faisant appel à une fonction d'interface de ⟨*mod*⟩.

La règle (3) restreint les principes d'écriture de fichiers en-tête énoncés en 5.2.2 (page 161). En particulier, le principe de masquage de l'implémentation implique la suppression des variables globales qui correspondent à une exportation d'implémentation. Ce principe ne peut pas toujours être appliqué strictement. En particulier, il s'avère parfois nécessaire de disposer de variables globales, visibles par plusieurs module, par exemple lors de la mise en œuvre d'une sauvegarde/restauration de contexte. Dans ce cas, on rappelle qu'une variable globale doit être définie dans un et un seul module, et simplement référencée dans les autres au moyen du mot clé `extern` (voir page 76). On considère alors que chaque variable globale **appartient** à un module unique, qui est celui dans lequel elle est définie. Seul le module propriétaire d'une variable globale a le droit de l'initialiser ou de la modifier; les autres modules se contentent de la consulter.

8.3.2 Un exemple plus complet

Mise en place du module "stat"

Cet exemple porte sur l'écriture d'une commande permettant d'effectuer des statistiques sur une liste de valeurs. Nous allons pour l'instant nous limiter

à trois traitements de base : le comptage du nombre de valeurs de la liste (option `-n`), le calcul de la somme de ces valeurs (option `-t`), et le calcul de leur moyenne (option `-m`). La commande, de nom `st`, lit la suite de nombres sur son entrée standard. Par défaut, toutes les options sont actives. Voici quelques exemples d'utilisation de cette commande.

– *Exemple 1*

```
$ st
1 10 100
1000 10000
C-d
   11111  2222.2       5
$
```

– *Exemple 2*

```
$ st -nt
1 10 100 1000 10000
C-d
        5   11111
$
```

– *Exemple 3*

```
$ ls -l stat.[ch]
-rw-r--r--  1 achille       376 Jun 13 02:30 stat.c
-rw-r--r--  1 achille       176 Jun 13 02:28 stat.h
$ lstaille -s stat.[ch]
     376
     176
$

$ ls -l *.[ch] */*.[ch] | wc -l
     316
$ lstaille -s *.[ch] */*.[ch] | st
  151662 479.943     316
$
```

Dans les deux premiers exemples, les données sont lues sur l'entrée standard. Dans le second, seules les options `n` et `t` sont actives. Le dernier exemple utilise le résultat de la commande `lstaille` dont le source est présenté en 11.1.2. Cette commande affiche sur sa sortie standard la taille de chacun des fichiers dont elle reçoit le nom en argument. On peut comparer son résultat avec celui produit par l'exécution de la ligne de commande `ls -l` sur un même jeu de données. La fin de l'exemple montre une combinaison de `lstaille` et de `stat` permettant d'obtenir une statistique sur la taille de tous les fichiers `.c` et `.h` du répertoire courant et de ses sous-répertoires (il s'agit des fichiers source listés dans cet ouvrage). Au moment de l'exécution de cet exemple, il y avait 316 fichiers, d'une taille moyenne de 480 caractères, pour une taille totale de 151662 caractères.

Le programme est organisé de la façon suivante :

– un fichier `st.c`, gérant la récupération des arguments et les affichages;

– un module stat effectuant les traitements statistiques.

L'interface du module stat se compose de cinq fonctions dont le prototype se trouve dans le fichier stat.h :

```
stat.h
#ifndef STAT_H
#define STAT_H

extern void   stat_initialiser(void);
extern void   stat_entrer(double);
extern int    stat_quantite(void);
extern double stat_total(void);
extern double stat_moyenne(void);

#endif  /* STAT_H */
stat.h
```

Ces fonctions servent respectivement à

– initialiser un nouveau calcul,
– entrer une nouvelle valeur dans la liste des nombres à analyser,
– obtenir le nombre de valeurs entrées,
– obtenir le total des valeurs,
– obtenir la moyenne des valeurs.

L'implémentation du module stat est simple, les calculs effectués ne nécessitant pas de mémoriser les valeurs entrées. L'environnement permanent se compose de deux variables de la liste :

– total : contenant la somme de toutes les valeurs entrées,
– nbr_valeurs : comptant le nombre de valeurs entrées.

```
stat.c
#include "stat.h"

static double total = 0.0;
static int nbr_valeurs = 0;

void stat_initialiser()
{
    total = 0.0;
    nbr_valeurs = 0;
}

void stat_entrer(double v)
{
    total += v;
    nbr_valeurs++;
}

int stat_quantite()
{
    return nbr_valeurs;
}
```

```
double stat_total()
{
    return total;
}

double stat_moyenne()
{
    return  total / (double) nbr_valeurs;
}
```

stat.c

Dans le fichier `st.c`, la récupération des arguments de la commande est effectuée au moyen de la fonction `extraire_options` (voir page 224). La seule difficulté réelle réside dans l'application du principe d'abstraction de constantes littérales (voir page 256). Nous avons regroupé dans un fichier de nom `st.def` toutes les définitions de constantes littérales susceptibles d'être modifiées, et fait en sorte que l'implémentation soit aussi insensible que possible à ces modifications.

st.def

```
/* Commandes */

#define COMMANDE_TOTAL       't'
#define COMMANDE_MOYENNE     'm'
#define COMMANDE_QUANTITE    'n'

/* Longueur du champ d'affichage */

#define LONGUEUR_AFFICHAGE   8
```

st.def

La déclaration

```
static char options_reconnues[] =
{
    COMMANDE_TOTAL, COMMANDE_MOYENNE,
    COMMANDE_QUANTITE, '\0'
};
```

permet de construire une chaîne formée des caractères d'option indépendamment de la valeur de ces caractères. La saisie des données est effectuée par la fonction `lire_donnees` qui itère l'appel à `stat_entrer` jusqu'à la détection d'une fin de fichier. L'affichage du résultat est effectué par la fonction `afficher_resultats` en fonction des options positionnées.

st.c

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

#include "options.h"
#include "stat.h"
#include "chaine.h"
```

```
#include "st.def"

static char options_reconnues[] =
{
    COMMANDE_TOTAL, COMMANDE_MOYENNE,
    COMMANDE_QUANTITE, '\0'
};

static int longueur_affichage = LONGUEUR_AFFICHAGE;
static char *argv0;

static void lire_donnees();
static void afficher_resultats();
static void usage();

main(int argc, char *argv[])
{
    char *options = extraire_options(argv+1,
                                     options_reconnues);

    argv0 = argv[0];
    if (options == NULL)
        usage();
    if (*options == '\0')
    {
        free(options);
        options = chaine_dup(options_reconnues);
    }

    lire_donnees();
    afficher_resultats(options);
    free(options);
}

static void lire_donnees()
{
    for (;;)
    {
        int n;

        if (scanf("%d", &n) == EOF)
            break;
        stat_entrer(n);
    }
}

static void afficher_resultats(char *options)
{
    while (*options != '\0')
    {
        switch (*options)
        {
          case COMMANDE_QUANTITE:
            printf("%*d", longueur_affichage, stat_quantite());
            break;

          case COMMANDE_TOTAL:
```

```
            printf("%*g", longueur_affichage, stat_total());
            break;

          case COMMANDE_MOYENNE:
            printf("%*g", longueur_affichage, stat_moyenne());
            break;

          default:
            fprintf(stderr, "%s:%c: option non implementee\n",
                    argv0, *options);
            break;
        }
        options++;
    }
    printf("\n");
}

static void usage()
{
    printf("Usage: %s -%s\n", argv0, options_reconnues);
    exit(1);
}
```

st.c

L'abstraction des constantes littérales permet des modifications très simples de leur définition. Pour associer le caractère `s` au calcul du total et afficher les résultats sur des champs de 12 caractères, il suffit de modifier le fichier `st.def` en

st.def

```
/* Commandes */

#define COMMANDE_TOTAL      's'
#define COMMANDE_MOYENNE    'm'
#define COMMANDE_QUANTITE   'n'

/* Longueur du champs d'affichage */

#define LONGUEUR_AFFICHAGE  12
```

st.def

et de recompiler l'application sans rien changer d'autre à son source. On peut vérifier qu'après cette modification, le comportement de la commande reste cohérent. En particulier, le message d'erreur `Usage:...` est cohérent.

```
$ st -t
Usage: st -smn
$ st -s -m
10 100
C-d
         110          55
$
```

Modification de l'implémentation

Supposons maintenant que l'on ait besoin d'étendre le module `stat` en ajoutant des traitements plus puissants (histogrammes, moyenne quadratique...). Ces calculs nécessitent de conserver la liste des valeurs entrées, et par conséquent, dans un premier temps, de modifier l'implémentation du module. Cette modification, qui est loin d'être mineure puisqu'elle nécessite une redéfinition complète des structures de données du module, peut-être effectuée sans changer une ligne de l'interface.

Voici la nouvelle implémentation du module `stat`. Les valeurs entrées sont mémorisées dans un vecteur dont la taille est étendue lorsque c'est nécessaire par un mécanisme d'allocation/rallongement Le calcul de la somme, qui est nécessaire à deux endroits différents du fichier, justifie l'écriture d'une fonction auxiliaire, `calcul_somme`, locale au module. On optimise les traitements en évitant de refaire ce calcul lorsqu'aucune nouvelle donnée n'a été entrée. Pour cela, on rajoute à l'environnement permanent du module la variable

```
static int total_valide = 0;
```

indiquant si le total courant est valide. Elle est positionnée à *faux* lors de chaque appel à `stat_entrer`, et à *vrai* lors d'un appel `calcul_somme`.

═══ `stat.c` *(nouvelle implémentation)* ═══

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

#include "ERREURS.h"
#include "exit_if.h"
#include "stat.h"

/* Nombre de valeur par bloc de rallongement */
#define NBVPBLOC  256
/* Taille d'un bloc de rallongement */
#define TBLOC     (NBVPBLOC*sizeof(double))

static double *valeurs = NULL;
static int    taille_de_valeurs = 0;

static int    total_valide = 0;

static double total     = 0.0;
static int    nbvaleurs = 0;

static void calcul_somme();

void stat_initialiser()
{
    if (valeurs != NULL)
    {
        free(valeurs);
        valeurs = NULL;
    }
    taille_de_valeurs = 0;
```

```
    total_valide = 0;
    total = 0.0;
    nbvaleurs = 0;
}

void stat_entrer(double v)
{
    if (nbvaleurs*sizeof(double) == taille_de_valeurs)
    {
        if (valeurs == NULL)
        {
            /* Creation du vecteur valeurs */
            valeurs = malloc(TBLOC);
            EXIT_IF(valeurs==NULL, ERR_MALLOC);
            taille_de_valeurs = TBLOC;
        }
        else
        {
            /* Extension du vecteur valeurs */
            double *valeurs = realloc(valeurs,
                                      taille_de_valeurs+TBLOC);
            EXIT_IF(valeurs==NULL, ERR_REALLOC);
            taille_de_valeurs += TBLOC;
        }
    }
    valeurs[nbvaleurs++] = v;
    total_valide = 0;
}

int stat_quantite()
{
    return nbvaleurs;
}

double stat_total()
{
    calcul_somme();
    return total;
}

double stat_moyenne()
{
    calcul_somme();
    return  total / (double) nbvaleurs;
}

static void calcul_somme()
{
    int i = 0;

    if (total_valide)
        return;

    total = 0;
    while (i < nbvaleurs)
        total += valeurs[i++];
}
```

═══════════ stat.c *(nouvelle implémentation)* ═══

La modification de l'implémentation du module **stat** n'ayant pas changé son comportement, elle n'introduit aucun problème dans les applications déjà écrites. En particulier, on peut recompiler la commande **st** avec les anciens fichiers **st.c** et **stat.h**. L'ancienne et la nouvelle commande ont des comportements identiques.

```
$ lstaille -s *.[ch] | st
111184 469.131     237
$ lstaille -s *.[ch] | st_ni
111184 469.131     237
$
```

Il est maintenant possible, dans un second temps, d'étendre l'interface du module **stat** en rajoutant de nouvelles fonctions. Dans ce cas encore, on préserve la *compatibilité ascendante*, c'est-à-dire la compatibilité du nouveau module avec les applications plus anciennes.

8.3.3 Limites de l'implémentation de modules par des fichiers

Le modèle présenté ici est, sous cette forme ou sous des formes voisines, très largement utilisé. Il se prête parfaitement à la définition de modules comme le module **stat**, définis de façon statique, et pour lesquels une seule occurrence de l'entité implémentée par le module est nécessaire.

Cependant, on a souvent besoin de modules permettant de gérer dynamiquement plusieurs occurrences d'une même entité. Par exemple, si on implémente une pile de la façon que nous venons de décrire, il n'est pas possible de disposer simultanément de plusieurs piles distinctes.

Pour pallier cet inconvénient, nous allons étendre ce modèle afin de permettre à un module d'*instancier* plusieurs occurrences de l'entité qu'il code. Cela revient à définir un style de programmation par objets.

8.4 Un modèle de programmation par objets

8.4.1 Quelques aspects de la programmation par objets

Nous avons exprimé, en conclusion de la section précédente, le souhait de pouvoir disposer, dans un programme, de plusieurs instances de l'entité implémentée par un module. Cette implémentation comporte deux parties bien distinctes :

- l'*environnement permanent*, composé d'un ensemble de variables, locales au module, et de durée de vie illimitée;
- le *code*, composé d'un ensemble de fonctions, globales s'il s'agit de fonctions d'interface et locales sinon, et ayant en charge la gestion de l'envi-

ronnement local.

Cette séparation entre le code et l'environnement d'un module facilite la gestion simultanée de plusieurs occurrences de l'entité implémentée. En effet, pour chaque nouvelle occurrence, il est seulement nécessaire de créer une copie de l'environnement. Toutes les occurrences d'une même entité existant à un moment donné partagent le même code.

Revenons encore une fois sur l'exemple du générateur d'entiers. Pour pouvoir disposer simultanément de plusieurs générateurs évoluant indépendamment les uns des autres il suffit, d'une part, de créer pour chaque nouveau générateur une copie du triplet de variables

```
valeur_initiale
valeur_courante
pas
```

et, d'autre part, de faire opérer les fonctions d'interface sur l'une ou l'autre de ces copies. Cette séparation en un environnement privée duplicable et un ensemble de fonctions opérant sur une ou l'autre des instances de cet environnement est à la base de la programmation par objets.

Nous allons maintenant donner quelques définitions. Il ne s'agit pas ici de définir formellement les concepts généraux de la programmation par objets, mais plutôt de fixer une terminologie cohérente, compatible avec celle communément admise en P.P.O., et permettant de décrire sans ambiguïté les constructions qui vont être effectuées.

Un **objet** est la donnée :

- d'un ensemble de variables, encore appelées **attributs**, ou **variables d'instances**, constituant son environnement;
- d'un ensemble de fonctions, encore appelées **méthodes**, implémentant les traitements spécifiques à cet objet.

On appellera **classe** la description d'une même famille d'objets. Une classe est la donnée :

- d'un ensemble de méthodes implémentant le comportement de ses objets;
- de la description d'un ensemble d'attributs décrivant leur environnement;
- d'un mécanisme de construction dynamique d'objets appelé mécanisme d'**instanciation**.

Les objets produits par instanciation à partir d'une classe *C* sont appelés des **instances** de *C*. Le mécanisme général de communication avec un objet est appelé **envoi de message**, et l'objet sur lequel s'applique l'envoi de message est appelé le **receveur**.

Un envoi de message à un objet *O* est l'activation d'une méthode de la classe de l'objet *O* sur l'objet *O* lui-même. Par exemple, si `generateur` est une classe décrivant les générateurs d'entiers, est si `g1` et `g2` sont deux instances de cette classe, l'envoi du message `aller_au_debut` à l'objet `g1` réinitialise les variables d'instance de `g1`, alors que l'envoi du message `suivant` à `g2` incrémente le compteur `valeur_courante` de l'objet `g2` et retourne sa valeur.

On peut, sur ce principe, donner une sémantique simple de l'envoi de message. Nous allons tout d'abord fixer la forme générale d'une méthode de la façon suivante :

> *Une* **méthode** *M de la classe C est une fonction dont le premier paramètre est la référence à un objet de C.*

Sous cette hypothèse, l'**envoi de message** peut se définir de la façon suivante :

> *Si O est une instance de la classe C, et si M est une méthode de cette classe, l'envoi du message M à l'objet O consiste à appeler la méthode M de la classe C avec comme premier paramètre l'objet O.*

Comme nous l'avons déjà dit, du point de vue de la méthode M, l'objet O est le receveur du message, ou encore l'**objet courant** dont la méthode manipule l'environnement.

Le mécanisme que nous venons de décrire, qui est un autre mécanisme d'encapsulation, n'est que l'un des aspects de la programmation par objets, l'autre étant l'**héritage**. En fait, si l'encapsulation par objets est assez facilement réalisable dans la plupart des langages de programmation, l'héritage est par contre spécifique à la famille des langages à objets, à laquelle le langage C n'appartient pas.

Sans rentrer dans les détails, disons simplement que l'héritage est un mécanisme permettant de définir une nouvelle classe par raffinement d'une classe déjà définie. Son but principal est la réutilisabilité. Dans le cas des langages à objets typés, comme le langage C++, l'héritage permet également la mise en œuvre du *polymorphisme*, que l'on peut décrire comme la possibilité de gérer de façon homogène une famille d'objets ayant une même interface et des implémentations différentes. En fait, bien que l'héritage ne soit pas explicitement implémenté en C, il est cependant possible de mettre en œuvre un mécanisme basé sur le même principe, et permettant une implémentation du polymorphisme.

8.4.2 Une implémentation des classes en C

Définition de l'interface

Nous allons considérer un nouvel exemple, qui est une variante des générateurs d'entiers. Il s'agit de la définition d'une *roue numérique*, c'est-à-dire d'un générateur parcourant indéfiniment une liste de longueur finie, en revenant sur le premier élément de la liste chaque fois que le dernier est atteint. Une roue numérique est définie par un couple d'entiers qui sont les deux valeurs extrêmes de la liste qu'elle génère. Ainsi la roue numérique définie par le couple $\prec 1, 5 \succ$ produit la suite infinie

```
1 2 3 4 5 1 2 3 4 5 1 2 ...
```

Commençons par considérer l'implémentation de l'environnement d'un objet de la classe `roue_num`. Une solution naturelle consiste à définir trois variables

```
valeur_initiale
valeur_courante
valeur_maximale
```

codant les bornes et la valeur courante de la roue. Cet environnement doit pouvoir être recopié à chaque instanciation d'un nouvel objet. Pour cela, on regroupe toutes les variables de l'environnement dans une structure dont on

alloue dynamiquement une copie à chaque instanciation. Nous allons donner à cette structure le nom de la classe :

```
struct roue_num
{
    int valeur_initiale;
    int valeur_courante;
    int valeur_maximale;
};
```

La structure `struct roue_num` définit le type d'un objet de la classe `roue_num`. Le langage C étant un langage typé, il est nécessaire d'exporter le type `struct roue_num` dans toute partie de programme où sont manipulés des objets de cette classe. Or cette définition concerne l'implémentation de l'objet, et si l'on veut respecter le principe de masquage de l'implémentation, elle doit être encapsulée dans la définition de la classe, et ne pas apparaître dans son interface.

Ces deux contraintes, à priori incompatibles, peuvent cependant être satisfaites simultanément. En fait, à l'extérieur d'une classe, on manipulera seulement des références aux objets de cette classe, et non les objets eux-mêmes. Il suffit donc de déclarer un objet de la classe `roue_num` avec le type :

```
struct roue_num *
```

Le compilateur C se satisfait de cette définition incomplète tant qu'il ne rencontre pas d'expression dont la traduction nécessite une connaissance du type `struct roue_num`, c'est-à-dire une expression comportant une indirection, effectuée par l'un des deux opérateurs `*` ou `->`, sur un objet de ce type. C'est précisément le but du masquage d'implémentation que d'interdire ce genre d'opérations.

Afin d'abstraire le plus possible la déclaration et la manipulation des objets, on définit au moyen de la construction `typedef` un nom pour le type `struct roue_num *`. Le plus naturel est d'utiliser le nom `roue_num` lui même :

```
typedef struct roue_num *roue_num;
```

Plus généralement, le type exporté d'un objet de la classe ⟨*cls*⟩ est :

`typedef struct` ⟨*cls*⟩ `*`⟨*cls*⟩`;`

Pour simplifier l'écriture de l'interface d'une classe, on introduit la définition de la pseudo-fonction `CLASSE` qui effectue automatiquement cette déclaration. Cette définition est placée dans le fichier `objet.h`, qui sera systématiquement inclus dans la description de l'interface de chaque classe.

```
═══════ objet.h ═══════════════════════
#ifndef OBJET_H
#define OBJET_H

#define CLASSE(c_ident)   typedef struct c_ident *c_ident

#endif  /* OBJET_H */
═══════════════════════ objet.h ═══════
```

Il est maintenant nécessaire de définir une fonction d'instanciation, recevant en paramètre les valeurs définissant la roue à instancier, et retournant une ins-

tance correctement initialisée. Nous avons choisi de donner à la fonction d'instanciation d'une classe ⟨*cls*⟩ le nom `FAIRE_`⟨*cls*⟩.[3] La fonction d'instanciation de la classe `roue_num` reçoit deux entiers en paramètres (les deux bornes de la suite) et retourne la référence à un objet de type `struct roue_num`. Avec les conventions adoptées précédemment, le prototype de cette fonction est le suivant :

```
roue_num FAIRE_roue_num(int, int);
```

La classe `roue_num` comporte trois méthodes :

- `initialiser` : qui repositionne la roue au début;
- `avancer` : qui fait avancer la roue d'une position;
- `valeur` : qui retourne la valeur courante de la roue.

Le premier paramètre de chaque méthode, le receveur du message, est une référence à un objet de la classe `roue_num`, c'est-à-dire un objet de type `roue_num`. Dans ce cas précis, les méthodes n'ont pas d'autre paramètre.

D'autre part, la convention de nommage des fonctions d'interfaces permettant d'éviter les collisions de noms de fonction est toujours valide. Par conséquent, chaque nom de méthode est préfixé par celui de sa classe. Le prototype complet des méthodes de la classe `roue_num` est :

```
void  roue_num_initialiser(roue_num);
int   roue_num_avancer(roue_num);
int   roue_num_valeur(roue_num);
```

Remarque 32 *La méthode* `roue_num_avancer` *est de type* `int` *car elle retourne un booléen qui vaut* vrai *sauf lorsque la roue repasse par sa valeur initiale.*

Il est maintenant possible de préciser, dans ce modèle, la définition de l'envoi de message :

> *L'***envoi du message** *⟨m⟩ à un objet ⟨o⟩ avec les paramètres ⟨e_1⟩, ..., ⟨e_k⟩, est l'évaluation de l'appel de fonction*
>
> ⟨*cls*⟩`_`⟨*m*⟩`(`⟨*o*⟩`,` ⟨e_1⟩`, ..., `⟨e_k⟩`)`
>
> *où ⟨cls⟩ est la classe de l'objet ⟨o⟩.*

Voici le source complet de l'interface de la classe `roue_num` :

```
roue_num.h
#ifndef ROUE_NUM_H
#define ROUE_NUM_H

#include "objet.h"

CLASSE(roue_num);

extern roue_num FAIRE_roue_num(int, int);
extern void     roue_num_initialiser(roue_num);
extern int      roue_num_avancer(roue_num);
extern int      roue_num_valeur(roue_num);
```

[3] En anglais, il est d'usage d'appeler `new`, ou `create`, la fonction d'instanciation. Ici, la fonction d'instanciation pourrait s'appeler `num_wheel_new`.

```
#endif  /* ROUE_NUM_H */
```
roue_num.h

Définition de l'implémentation

Nous allons maintenant aborder l'écriture du fichier `roue_num.c` définissant l'implémentation de la classe. Celui-ci contient tout d'abord la définition de la structure `struct roue_num` établie plus haut :

```
struct roue_num
{
    int valeur_initiale;
    int valeur_courante;
    int valeur_maximale;
};
```

Il contient ensuite la définition de la fonction d'instanciation `FAIRE_roue_num`, et celle des trois méthodes. La fonction d'instanciation se décompose en trois parties :

- allocation dynamique d'une zone mémoire de taille

  `sizeof(struct roue_num)`
- initialisation des variables d'instance, c'est-à-dire des champs de la structure allouée;
- retour de la référence à cette structure.

Afin de limiter le nombre des définitions, nous avons choisi de noter uniformément l'objet manipulé par défaut par la fonction d'instanciation et par les méthodes. Il s'agit de la référence à l'objet instancié dans le premier cas, et de la référence au receveur du message dans le second. Dans les deux cas, on le notera `recv`.

[4] Compte tenu de cette convention d'écriture, l'allocation de l'environnement d'un objet de la classe `roue_num` s'écrit :

```
roue_num recv = malloc(sizeof(struct roue_num));
```

De manière générale, l'allocation de la structure codant un objet de la classe ⟨*cls*⟩ s'écrit :

```
⟨cls⟩ recv = malloc(sizeof(struct ⟨cls⟩));
```

Pour faciliter l'écriture des classes, nous utiliserons une pseudo-fonction `INSTANCIER` recevant en argument le nom de la classe et générant le code de l'allocation. Cette définition est contenue dans le fichier `classe.h` :

classe.h
```
#ifndef CLASSE_H
#define CLASSE_H

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

[4]En anglais, il est courant de noter cette variable `self` ou `this`.

```
#include "exit_if.h"

#define INSTANCIER(c_ident)    \
    c_ident recv = malloc(sizeof(struct c_ident));\
    EXIT_IF(recv==NULL, "erreur instanciation: " #c_ident)

#endif  /* CLASSE_H */
```
classe.h

On peut remarquer l'utilisation des mécanismes ANSI de génération de chaîne par le préprocesseur (voir page 158) et de concaténation de constantes littérales chaînes par le compilateur, afin de produire un message d'erreur d'instanciation incluant le nom de la classe dans laquelle l'erreur se produit. On peut tester le fonctionnement on moyen du programme de test suivant :

test_instancier.c

```
#include "objet.h"
#include "classe.h"
#include "unites.h"

CLASSE(trop_grand);

struct trop_grand
{
    char vecteur[100*MEGA];
};

main()
{
    INSTANCIER(trop_grand);
}
```
test_instancier.c

Il définit un objet de taille 100 MO, ce qui provoque une erreur d'allocation dynamique lors de l'instanciation :

```
$ test_instancier
>EXIT:test_instancier.c:14: erreur instanciation: trop_grand
$
```

La définition d'une méthode s'effectue par référence au receveur. Prenons l'exemple de la méthode `roue_num_initialiser`. Son travail consiste à affecter à l'attribut `valeur_courante` de l'environnement du receveur la valeur de l'attribut `valeur_initiale` de ce même environnement. Cette opération s'écrit tout simplement

```
recv->valeur_courante = recv->valeur_initiale;
```

Voici pour finir le source complet de l'implémentation de la classe `roue_num`. L'inclusion du fichier en-tête `roue_num.h` dans le fichier source `roue_num.c` permet au compilateur de vérifier la concordance entre interface et implémentation.

roue_num.c

```
#include <stdio.h>

#include "classe.h"
#include "roue_num.h"

struct roue_num
{
    int valeur_initiale;
    int valeur_courante;
    int valeur_maximale;
};

roue_num FAIRE_roue_num(int valeur_initiale,
                        int valeur_maximale)
{
    INSTANCIER(roue_num);

    recv->valeur_initiale = valeur_initiale;
    recv->valeur_maximale = valeur_maximale;
    roue_num_initialiser(recv);
    return recv;
}

void roue_num_initialiser(roue_num recv)
{
    recv->valeur_courante = recv->valeur_initiale;
}

int roue_num_avancer(roue_num recv)
{
    if (recv->valeur_courante == recv->valeur_maximale)
    {
        roue_num_initialiser(recv);
        return 0;
    }
    else
    {
        (recv->valeur_courante)++;
        return 1;
    }
}

int roue_num_valeur(roue_num recv)
{
    return recv->valeur_courante;
}
```

roue_num.c

Voici maintenant un exemple d'utilisation de cette classe. Il s'agit d'une commande gérant la rotation de deux roues identiques, mais tournant à des vitesses différentes. Les deux roues sont numérotées à partir de zéro. La taille et la vitesse de chaque roue sont passées, dans cet ordre, en arguments à la commande. Le programme s'arrête lorsque les deux roues passent simultanément par la même valeur. Voici un premier exemple d'exécution avec deux

roues de taille 10, et de vitesses respectives 1 et 2.

```
$ deux_roues 10 1 2

Depart avec
  [  0 ][  0 ]
Rotations
  [  1 ][  2 ]
  [  2 ][  4 ]
  [  3 ][  6 ]
  [  4 ][  8 ]
  [  5 ][  0 ]
  [  6 ][  2 ]
  [  7 ][  4 ]
  [  8 ][  6 ]
  [  9 ][  8 ]
Arret avec
  [  0 ][  0 ]

$
```

La première roue défile un par un les entiers de 0 à 9, tandis que la seconde avance de deux en deux. sur cet exemple, le programme s'arrête lorsque les deux roues passent simultanément par 0. Voici d'autres exemples d'exécution de cette commande :

```
$ deux_roues 10 1 5

Depart avec
  [  0 ][  0 ]
Rotations
  [  1 ][  5 ]
  [  2 ][  0 ]
  [  3 ][  5 ]
  [  4 ][  0 ]
Arret avec
  [  5 ][  5 ]

$ deux_roues 100 24 34

Depart avec
  [  0 ][  0 ]
Rotations
  [ 24 ][ 34 ]
  [ 48 ][ 68 ]
  [ 72 ][  2 ]
  [ 96 ][ 36 ]
  [ 20 ][ 70 ]
  [ 44 ][  4 ]
  [ 68 ][ 38 ]
  [ 92 ][ 72 ]
  [ 16 ][  6 ]
Arret avec
```

```
   [ 40 ][ 40 ]
 $
```

La commande `deux_roues` est organisée en trois parties : la récupération des arguments, l'instanciation de deux roues numériques, et la gestion de la rotation des roues. L'instanciation d'une roue $\langle r \rangle$ de bornes $\prec b_1, b_2 \succ$ s'écrit simplement

`roue_num` $\langle r \rangle$ `= FAIRE_roue_num(`b_1`,` b_2`);`

La commande utilise deux fonctions locales : `avancer` faisant avancer une roue de plusieurs positions, et `afficher` effectuant l'affichage des valeurs des deux roues.

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

#include "roue_num.h"

static void afficher(char*, int, int);
static void avancer(roue_num, int);

main(int argc, char *argv[])
{
    /* Recuperation des arguments:
       taille, vitesse 1, vitess 2 */
    int maximum = argc >= 2 ? atoi(argv[1])-1 : 2;
    int vitesse_1 = argc >= 3 ? atoi(argv[2]) : 1;
    int vitesse_2 = argc >= 4 ? atoi(argv[3]) : 1;

    /* Instanciation des deux roues */
    roue_num r1 = FAIRE_roue_num(0, maximum);
    roue_num r2 = FAIRE_roue_num(0, maximum);

    /* Gestion de la rotation des roues */
    afficher("\nDepart avec\n  ", roue_num_valeur(r1),
             roue_num_valeur(r2));
    printf("Rotations\n");
    for (;;)
    {
        avancer(r1, vitesse_1);
        avancer(r2, vitesse_2);
        if (roue_num_valeur(r1) == roue_num_valeur(r2))
            break;
        afficher("  ", roue_num_valeur(r1),
                 roue_num_valeur(r2));
    }
    afficher("Arret avec\n  ", roue_num_valeur(r1),
             roue_num_valeur(r2));
    printf("\n");
}

static void avancer(roue_num r, int vitesse)
{
```

```
    while (vitesse > 0)
    {
        roue_num_avancer(r);
        vitesse--;
    }
}

static void afficher(char *s, int r1, int r2)
{
    printf("%s[ %2d ][ %2d ]\n", s, r1, r2);
}
```

deux_roues.c

8.4.3 Gestion de la mémoire

Le modèle que nous venons de définir offre un mécanisme d'instanciation d'objets et une mise en œuvre de l'envoi de message. Par contre, il ne permet pas de libérer les objets inutiles. En effet, la solution consistant à libérer un objet ⟨*ident*⟩ par un appel direct à la fonction `free` de libération de la mémoire :

`free(`⟨*ident*⟩`)`

est à rejeter pour deux raisons. La première est d'ordre technique : la fonction d'instanciation a très bien pu être implémentée avec une autre fonction d'allocation dynamique que celle de la bibliothèque standard. La seconde raison est plus générale. L'environnement d'un objet peut comporter, en plus de la structure codant le type de l'objet, des données allouées dynamiquement (chaîne caractères, liste...). Dans ce cas, un appel à la fonction `free` sur la référence à l'objet détruira seulement cette structure, et toutes les zones mémoire qu'elle référençait seront perdues. C'est pour gérer ce problème qu'est intégré, dans plusieurs langages de programmation par objets, un mécanisme de *ramasse-miettes*, analogue à celui que l'on trouve dans les langages fonctionnels, et gérant la récupération automatique des zones mémoire déréférencées. Ce mécanisme n'étant pas implémenté en C, à moins de récrire sa propre gestion de mémoire,[5] il est nécessaire de munir toute classe ⟨*cls*⟩ d'un mécanisme de libération défaisant le travail effectué lors de l'instanciation. Nous avons choisi de noter cette fonction `DEFAIRE_`⟨*cls*⟩.[6]

Nous allons développer ce point en considérant l'implémentation de la classe `c_anneau` d'un anneau de caractères. Il s'agit d'une zone mémoire, de taille fixe, contenant des caractères, et organisée de façon circulaire (voir figure 8.2). Tant que l'anneau n'est pas plein, tout nouveau caractère est ajouté à la fin (la *tête*) de l'anneau. L'anneau est de plus muni d'un curseur de lecture, appelé `position`, utilisé pour parcourir le contenu de l'anneau. On définit trois méthodes :

- `c_anneau_ajouter` : ajout d'un caractère à la fin de l'anneau;
- `c_anneau_aller_au_dernier` : positionnement du curseur de lecture sur le dernier élément entré;

[5]Ce qui n'est pas nécessairement une solution déraisonnable.

[6]En anglais, cette fonction est généralement appelée `delete`.

- `c_anneau_precedent` : recul du curseur d'une position et retour du caractère pointé

Lorsque le curseur arrive au début de l'anneau, un nouvel appel à la méthode `precedent` le place sur le caractère de tête. L'interface de cette classe est la suivante :

```
c_anneau.h
#ifndef C_ANNEAU_H
#define C_ANNEAU_H

#include "objet.h"

CLASSE(c_anneau);

extern c_anneau FAIRE_c_anneau(int);
extern void     DEFAIRE_c_anneau(c_anneau);
extern void     c_anneau_ajouter(c_anneau, char);
extern void     c_anneau_aller_au_dernier(c_anneau);
extern char     c_anneau_precedent(c_anneau);

#endif  /* C_ANNEAU_H */
c_anneau.h
```

On peut remarquer la présence de la fonction de libération `DEFAIRE_anneau` évoquée précédemment.

Nous allons implémenter cette structure de donnée au moyen de cinq variables codant l'état de l'anneau, et d'un vecteur de caractères codant son contenu (voir figure 8.3). Ces cinq variables sont :

- `debut` : un pointeur vers le début du vecteur codant le contenu de l'anneau;
- `taille` : la taille du vecteur (c'est la taille de l'anneau);
- `tete` : un pointeur sur la position à laquelle se fera le prochain ajout de caractère;
- `plein` : un booléen indiquant si l'anneau est plein, et dont nous allons voir l'utilité plus loin;
- `position` : un pointeur codant la position du curseur de lecture.

Lorsque l'anneau est plein, c'est-à-dire lorsque le pointeur `tete` dépasse la fin du vecteur, ce dernier est ramené au début de façon à écraser les caractères les plus anciens. De cette façon, on dispose toujours des n derniers caractères entrés, n étant la taille de l'anneau. La figure 8.4 représente l'anneau plein et la figure 8.5 son implémentation. Nous avons tous les éléments pour réaliser l'implémentation de la fonction d'instanciation et des méthodes. Considérons maintenant la fonction de libération `DEFAIRE_anneau`. Il s'agit d'une méthode particulière[7], ne retournant rien, et libérant toute la place mémoire utilisée pour implémenter l'environnement du receveur. Après son exécution, l'objet sur lequel portait l'envoi de message est indéfini. Elle s'écrit :

[7] Pour respecter les conventions de notation, cette méthode devrait s'appeler `c_anneau_defaire`. Du moment que cela n'introduit pas de problème de collision de nom de fonction, nous avons préféré forger son nom sur le même principe que la fonction d'instanciation.

```
void DEFAIRE_c_anneau(c_anneau recv)
{
    free(recv->debut);
    free(recv);
}
```

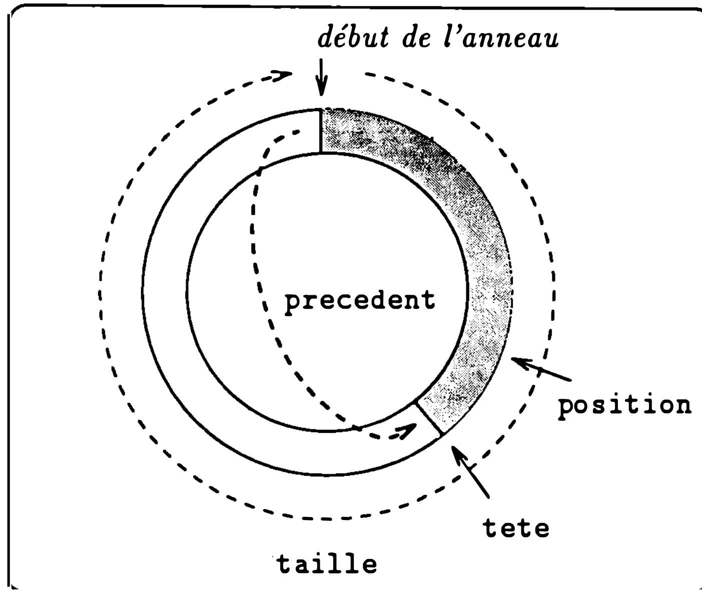


Figure 8.2: Modèle de l'anneau de caractères (non plein)

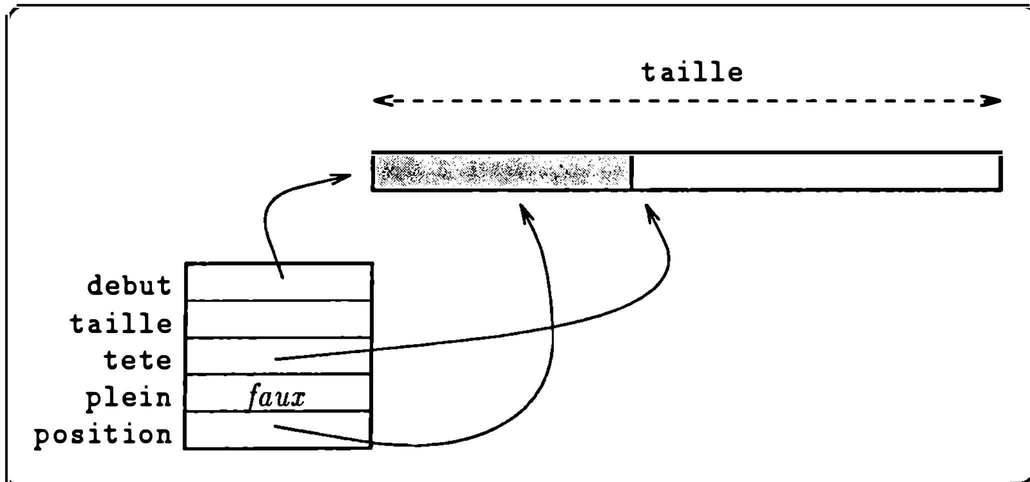


Figure 8.3: Implémentation de l'anneau (non plein)

Dans le cas de la classe `roue_num` de l'exemple précédent, cette méthode est simplement

```
voide DEFAIRE_roue_num(roue_num recv)
{
    free(recv);
}
```

L'implémentation des différentes méthodes de la classe `c_anneau` n'est pas très difficile. La méthode `c_anneau_ajouter` gère l'incrémentation du pointeur tête. Lorsqu'il atteint la fin du vecteur, c'est-à-dire lorsque

```
recv->tete == recv->debut + recv->taille
```

le pointeur `tete` est ramené au début et le booléen `plein` est positionné à *vrai*.

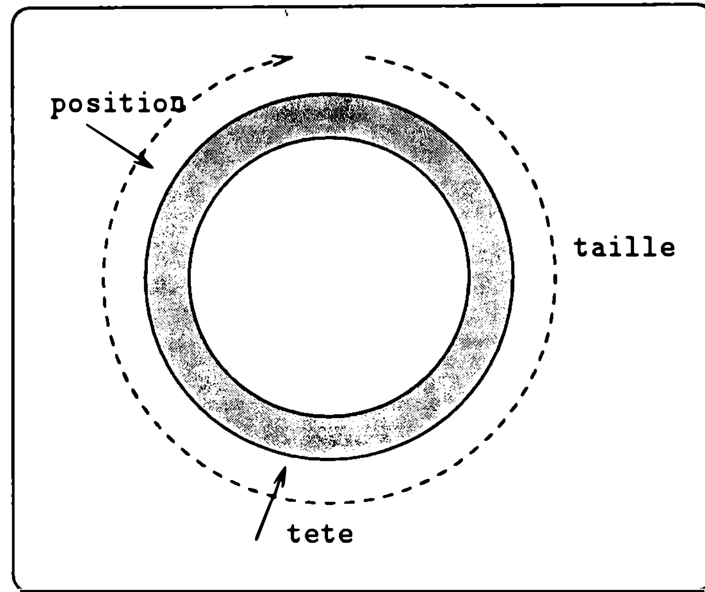


Figure 8.4: Modèle de l'anneau de caractères (plein)

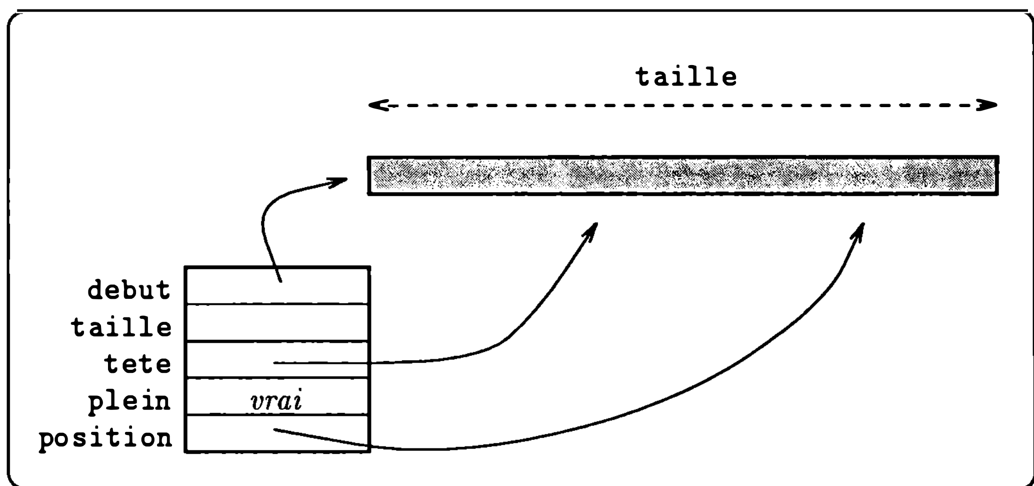


Figure 8.5: Implémentation de l'anneau de caractères (plein)

La méthode `c_anneau_precedent` gère le recul du curseur de lecture. Lorsque celui est positionné au début du vecteur, le traitement diffère selon que l'anneau est plein ou non. Dans le premier cas, il est positionné sur le dernier caractère du vecteur, et sur le caractère de tête dans le second cas.

c_anneau.c

```
#include <assert.h>

#include "classe.h"
#include "c_anneau.h"

struct c_anneau
{
    char *debut;
    int  taille;
    char *tete;
    int  plein;
    char *position;
};

c_anneau FAIRE_c_anneau(int taille)
{
    INSTANCIER(c_anneau);

    recv->tete = recv->debut = malloc(taille*sizeof(char));
    recv->taille = taille;
    recv->plein = 0;
    return recv;
}

void DEFAIRE_c_anneau(c_anneau recv)
{
    free(recv->debut);
    free(recv);
}

void c_anneau_ajouter(c_anneau recv, char c)
{
    assert(recv->tete < recv->debut + recv->taille);

    *(recv->tete)++ = c;
    if (recv->tete == recv->debut + recv->taille)
    {
        recv->tete = recv->debut;
        recv->plein = 1;
    }
}

void c_anneau_aller_au_dernier(c_anneau recv)
{
    recv->position = recv->tete;
}

char c_anneau_precedent(c_anneau recv)
{
    if (recv->position > recv->debut)
        (recv->position)--;
    else if (recv->plein)
        recv->position = recv->debut + recv->taille -1;
    else if (recv->tete > recv->debut)
        recv->position = recv->tete - 1;
    else
        /* Anneau vide */
```

```
        return '\0';
    return *(recv->position);
}
```

═══════════ c_anneau.c ═══════════

Nous allons tester cette classe au moyen d'un programme interactif traitant les commandes suivantes :

– n : libération de l'anneau courant et instanciation d'un nouvel anneau;

– > : positionnement du curseur de lecture à la fin de l'anneau;

– < : affichage du caractère courant et recul du curseur de lecture;

Tout autre caractère saisi est entré dans l'anneau. Nous allons effectuer le test avec un anneau de 6 caractères. Voici un exemple commenté de session sous cette commande :

```
$ test_anneau

[1]: 1234

[2]: > <<<<<<
(4)(3)(2)(1)(4)(3)
[3]: 567

[4]: <<<<<<<<
(2)(7)(6)(5)(4)(3)(2)(7)
[5]: n

[6]: <
()
[7]: 123456789 > <<<<<<<<<
(9)(8)(7)(6)(5)(4)(9)(8)(7)
[8]: C-d
$
```

Le prompt affiché par `test_anneau` contient le numéro de la ligne de commande saisie. La première ligne saisie provoque l'entrée dans l'anneau des caractères 1, 2, 3 et 4. La seconde provoque le positionnement du curseur à la fin de l'anneau suivi de l'itération de six appels à `c_anneau_precedent`. On vérifie qu'on parcourt l'anneau circulairement à partir de la fin. La troisième ligne provoque l'entrée des caractères 5, 6 et 7. La taille de l'anneau étant de 6 caractères, le septième caractère entré remplace le premier, ce que l'on vérifie avec la quatrième ligne. La cinquième ligne provoque la libération de l'anneau et l'instanciation d'un nouvel anneau vide. Les lignes suivantes testent le fonctionnement du nouvel anneau.

Cette commande utilise, pour générer le prompt, le module implémentant un générateur élémentaire d'entiers présenté au début de ce chapitre.

═══════════ test_anneau.def ═══════════

```
#define NOUVEL_ANNEAU       'n'
```

```
#define ALLER_EN_FIN       '>'
#define AFFICHER_PRECEDENT '<'
```

═══════════════════ test_anneau.def ════

════ test_anneau.c ═══════════════════

```
#include <stdio.h>

#include "generateur.h"
#include "c_anneau.h"

#include "test_anneau.def"

#define TAILLE_ANNEAU 6

static void prompt();

main()
{
    c_anneau anneau = FAIRE_c_anneau(TAILLE_ANNEAU);

    prompt();
    for (;;)
    {
        int c = getchar();

        if (c == EOF)
            break;

        switch (c)
        {
          case  NOUVEL_ANNEAU :
            DEFAIRE_c_anneau(anneau);
            anneau = FAIRE_c_anneau(TAILLE_ANNEAU);
            break;

          case ALLER_EN_FIN:
            c_anneau_aller_au_dernier(anneau);
            break;

          case AFFICHER_PRECEDENT:
            printf("(%c)", c_anneau_precedent(anneau));
            break;

          default:
            c_anneau_ajouter(anneau, c);
            break;

          case '\n':
            prompt();
            /* NOBREAK */
          case ' ':
          case '\t':
            break;
        }
    }
    printf("\n");
```

```
}

static void prompt()
{
    printf("\n[%d]: ", generateur());
}
```
test_anneau.c

8.5 Paramétrer pour réutiliser

L'encapsulation des données et le masquage de l'implémentation favorisent, certes, la réutilisabilité, mais ils ne résolvent pas tous les problèmes. Pour accroître les possibilités de réutilisation d'un module, il est nécessaire de paramétrer autant que possible ses fonctionnalités.

Une solution classique consiste à définir un ensemble de comportements, *câblés* dans la classe, et recouvrant un maximum de cas. La sélection d'un comportement particulier est effectuée, au moyen d'un aiguillage, en fonction d'un choix passé en paramètre.

Si cette solution est classique, c'est plus pour la simplicité de sa mise en œuvre que pour son efficacité réelle. En effet, prévoir de cette manière tous les comportements possibles d'une classe reviendrait tout simplement à implémenter dans cette classe un mécanisme complet d'interprétation d'un langage de programmation.

Encore une fois, à cette approche dirigée par les traitements s'oppose une approche dirigée par les données, consistant à transmettre en paramètre non plus la codification d'un traitement, mais le traitement lui-même, ou plus exactement un pointeur vers la fonction qui le réalise. Nous allons développer cette approche en étudiant l'écriture d'une nouvelle classe utilisant la classe `roue_num`. Il s'agit de la classe `totalisateur`, implémentant le tableau d'affichage d'un compteur, composé d'une suite de roues numériques. On peut voir sur la figure 8.6 la représentation d'un totalisateur composé de trois roues numériques.

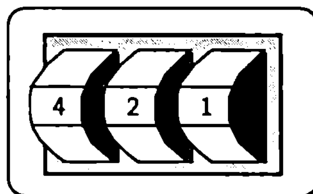


Figure 8.6: Un totalisateur composé de 3 roues numériques

Le principe du fonctionnement d'un totalisateur est le suivant :

1) Les roues du totalisateur sont ordonnées.
2) Une requête d'initialisation provoque la réinitialisation de toutes les roues.
3) Une requête d'incrémentation fait avancer la première roue d'une position.
4) Lorsqu'une roue repasse par sa position initiale, la roue suivante avance d'une position.

La dernière règle a pour but la gestion des retenues.

Commençons par définir l'interface d'un totalisateur afin de fixer son comportement.

```
══════ totalisateur.h ══════════════════════════
#ifndef TOTALISATEUR_H
#define TOTALISATEUR_H

#include "objet.h"

#include "roue_num.h"

CLASSE(totalisateur);

extern totalisateur FAIRE_totalisateur(void);
extern void   DEFAIRE_totalisateur(totalisateur);
extern void   totalisateur_ajouter_roue(totalisateur, roue_num);
extern void   totalisateur_initialiser(totalisateur);
extern void   totalisateur_avancer(totalisateur);
extern int    totalisateur_nombre_de_roues(totalisateur);
extern void   totalisateur_afficher(totalisateur, void (*)(int));

#endif  /* TOTALISATEUR_H */
══════════════════════════ totalisateur.h ══════
```

Lors de l'instanciation, le totalisateur est défini sans roue. Comme son nom l'indique, la méthode `totalisateur_ajouter_roue` permet d'ajouter une nouvelle roue au totalisateur. Les roues sont chaînées entre elles en commençant par la dernière, c'est-à-dire celle de poids fort. Chaque nouvelle roue est ajoutée en tête de liste.

La méthode `totalisateur_initialiser` remet toutes les roues à leur position d'origine, et la méthode `totalisateur_avancer` fait avancer d'une position la première roue. Bien entendu, si celle-ci repasse par sa position d'origine, c'est-à-dire si elle retourne la valeur *faux*, le totalisateur doit gérer le transport de la retenue et faire avancer la seconde roue, puis la troisième si la seconde repasse elle aussi par son origine, et ainsi de suite jusqu'à ce qu'une roue retourne *vrai* ou jusqu'à ce que toutes les roues aient été parcourues.

Nous allons implémenter l'environnement du totalisateur au moyen d'une liste de roues. Le chaînage est effectué à l'aide de la structure

```
struct emplacement
{
    struct emplacement *suivant;
    roue_num roue;
};
```

définie localement dans l'implémentation. La structure décrivant un totalisateur comporte le nombre de roues du totalisateur et un pointeur vers le premier emplacement de roue. Cette organisation est décrite sur la figure 8.7, avec le codage d'un totalisateur à trois roues.

L'instanciation d'un totalisateur consiste à initialiser une liste vide. La fonction de libération est un peu plus délicate. Elle consiste à parcourir la liste et, pour chaque emplacement, à libérer la roue référencée en utilisant la fonction `DEFAIRE_roue_num` puis l'emplacement lui-même. L'ajout d'une nouvelle roue

se décompose de la façon suivante :

- allocation dynamique d'un emplacement E,
- chaînage de la nouvelle roue sur E,
- chaînage de E sur le totalisateur,
- incrémentation du nombre de roues.

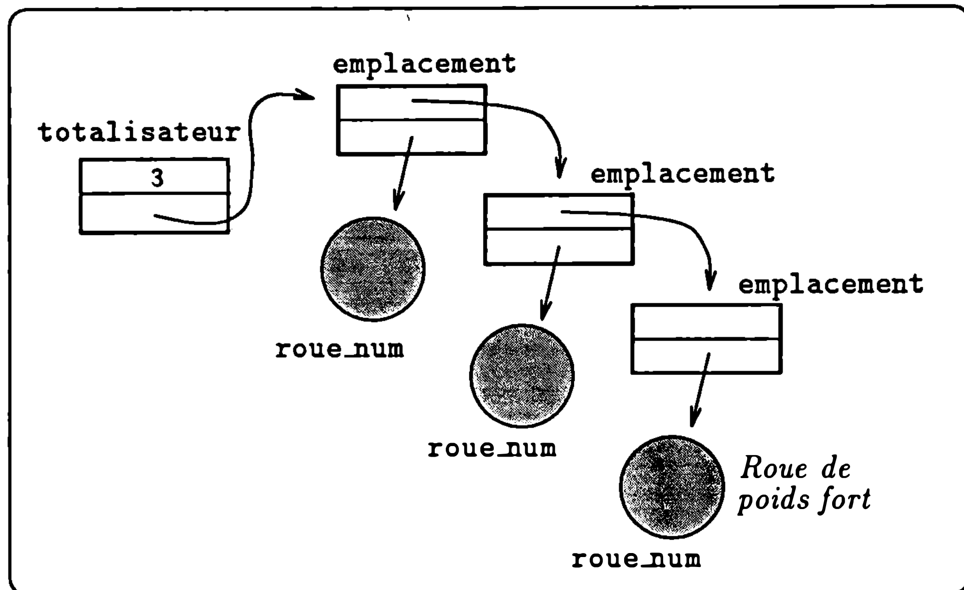


Figure 8.7: Codage d'un totalisateur par chaînage de roues (a)

L'initialisation consiste à parcourir la liste au moyen d'une itération en envoyant le message `initialiser` à chaque roue, et l'incrémentation du totalisateur à parcourir les roues en envoyant le message `avancer`, tant qu'aucune roue ne retourne *vrai*.

Le seul véritable problème qui se pose pour implémenter cette classe est lié à l'affichage. En effet, si l'on fixe le format de l'affichage dans l'implémentation du totalisateur, il y a peu de chance que l'on puisse réutiliser cette classe pour une nouvelle application.

Une solution consiste à passer en paramètre à `totalisateur_afficher` la fonction à appeler pour effectuer l'affichage. Vu depuis la classe `totalisateur`, l'affichage consiste à parcourir les roues en appelant la fonction d'affichage avec en paramètre la valeur courante de la roue. Le prototype de cette fonction est

`void` $\langle f \rangle$`(int)`

et celui d'une référence à cette fonction

`void (*)(int)`

Cette construction est explicitée en 2.3 (page 55).

⟹ `totalisateur.c` *(avec liste de roues)*
```
#include <stdio.h>
#include <stdlib.h>

#include "exit_if.h"
```

```
#include "classe.h"

#include "totalisateur.h"

struct emplacement
{
    struct emplacement *suivant;
    roue_num roue;
};

struct totalisateur
{
    int nombre_de_roues;
    struct emplacement *premier_emplacement;
};

static void afficher(struct emplacement *, void (*)(int));

totalisateur FAIRE_totalisateur()
{
    INSTANCIER(totalisateur);

    recv->nombre_de_roues = 0;
    recv->premier_emplacement = NULL;
    return recv;
}

void DEFAIRE_totalisateur(totalisateur recv)
{
    struct emplacement *p = recv->premier_emplacement;

    while (p != NULL)
    {
        struct emplacement *q = p;

        p = p->suivant;
        DEFAIRE_roue_num(q->roue);
        free(q);
    }
    free(recv);
}

void totalisateur_ajouter_roue(totalisateur recv, roue_num r)
{
    struct emplacement *p = malloc(sizeof(struct emplacement));
    EXIT_IF(p==NULL, "erreur allocation dynamique");

    p->roue = r;
    p->suivant = recv->premier_emplacement;
    recv->premier_emplacement = p;
    (recv->nombre_de_roues)++;
}

void totalisateur_initialiser(totalisateur recv)
{
    struct emplacement *p =  recv->premier_emplacement;

    while (p != NULL)
```

```
    {
        roue_num_initialiser(p->roue);
        p = p->suivant;
    }
}

void totalisateur_avancer(totalisateur recv)
{
    struct emplacement *p =  recv->premier_emplacement;

    while (p != NULL)
    {
        if (roue_num_avancer(p->roue))
            break;
        p = p->suivant;
    }
}

int totalisateur_nombre_de_roues(totalisateur recv)
{
    return recv->nombre_de_roues;
}

void totalisateur_afficher(totalisateur recv, void (*f)(int))
{
    afficher(recv->premier_emplacement, f);
}

static void afficher(struct emplacement *p, void (*f)(int))
{
    if (p != NULL)
    {
        afficher(p->suivant, f);
        f(roue_num_valeur(p->roue));
    }
}
```

totalisateur.c *(avec liste de roues)*

La fonction auxiliaire `afficher` est utilisée par `totalisateur_afficher` pour parcourir récursivement les roues à partir de la roue de poids fort. Il est maintenant très simple de fabriquer des totalisateurs avec un nombre quelconque de roues numériques, elles-mêmes de bornes quelconques. Par exemple, le programme `compteur` suivant construit un totalisateur fait de roues numérotées de zéro à neuf. Le nombre de roues et le nombre d'incrémentations à effectuer sont passés en arguments. La fonction d'affichage est ici très simple, mais il pourrait très bien s'agir d'une fonction graphique dessinant une représentation des roues et du cadran comme celle de la figure 8.6. Cela nécessiterait d'effectuer un traitement spécifique à chaque roue, mais nous verrons dans l'exemple suivant que cela ne pose pas de réel problème.

compteur.c

```
#include <stdlib.h>

#include "pluriel.h"
```

```
#include "roue_num.h"
#include "totalisateur.h"

static void afficher(totalisateur);
static void f(int );

main(int argc, char *argv[])
{
    int i;
    int nbr = (argc >= 2 ? atoi(argv[1]) : 1);
    int nbi = (argc >= 3 ? atoi(argv[2]) : 1);
    totalisateur compteur = FAIRE_totalisateur();


    for (i=0; i<nbr; i++)
    {
        roue_num r = FAIRE_roue_num(0,9);
        totalisateur_ajouter_roue(compteur, r);
    }
    totalisateur_initialiser(compteur);

    for (i = 0; i < nbi; i++)
    {
        totalisateur_avancer(compteur);
        afficher(compteur);
    }
}


static void afficher(totalisateur c)
{
    printf("  [ ");
    totalisateur_afficher(c, f);
    printf("]\n");
}

static void f(int v)
{
    printf("<%d> ", v);
}
```

compteur.c

Voici quelques exemples d'exécution de cette commande.

```
$ compteur 3 1001
  [ <0> <0> <1> ]
  [ <0> <0> <2> ]
  [ <0> <0> <3> ]
  [ <0> <0> <4> ]
  [ <0> <0> <5> ]
  [ <0> <0> <6> ]
  [ <0> <0> <7> ]
  [ <0> <0> <8> ]
  [ <0> <0> <9> ]
  [ <0> <1> <0> ]
```

```
   [ <0> <1> <1> ]
   [ <0> <1> <2> ]
     .
     .
     .
   [ <9> <9> <8> ]
   [ <9> <9> <9> ]
   [ <0> <0> <0> ]
   [ <0> <0> <1> ]
$
$  compteur 5 99999
   [ <0> <0> <0> <0> <1> ]
   [ <0> <0> <0> <0> <2> ]
   [ <0> <0> <0> <0> <3> ]
   [ <0> <0> <0> <0> <4> ]
     .
     .
     .
   [ <0> <7> <2> <9> <8> ]
   [ <0> <7> <2> <9> <9> ]
   [ <0> <7> <3> <0> <0> ]
   [ <0> <7> <3> <0> <1> ]
     .
     .
     .
   [ <9> <9> <9> <9> <6> ]
   [ <9> <9> <9> <9> <7> ]
   [ <9> <9> <9> <9> <8> ]
   [ <9> <9> <9> <9> <9> ]
$
```

Ce paramétrage de l'affichage du totalisateur est très souple d'utilisation. Voici, par exemple, un programme produisant des plaques minéralogiques. Limitons-nous, pour simplifier, à des plaques à trois chiffres et deux lettres. Il ne serait pas très difficile de traiter le cas général, avec ajout automatique d'une roue lors d'un dépassement de capacité.

Le totalisateur `plaque` est construit avec deux roues variant du caractère `'A'` au caractère `'Z'`, et trois roues variant du caractère `'0'` au caractère `'9'`. La fonction d'affichage est un peu plus sophistiquée que dans le cas précédent, du fait que les roues sont affichée dans un ordre différent de celui de la liste. Elle utilise un générateur d'entiers pour calculer la position des lettres à écrire.

```
========== plaque.c ==========
#include <stdlib.h>

#include "pluriel.h"
#include "generateur.h"

#include "roue_num.h"
#include "totalisateur.h"

static void afficher(totalisateur);
```

```
static void f(int );

main(int argc, char *argv[])
{
   int i;
   int nbi = (argc >= 2 ? atoi(argv[1]) : 1);
   totalisateur plaque = FAIRE_totalisateur();

   totalisateur_ajouter_roue(plaque, FAIRE_roue_num('A','Z'));
   totalisateur_ajouter_roue(plaque, FAIRE_roue_num('A','Z'));
   totalisateur_ajouter_roue(plaque, FAIRE_roue_num('0','9'));
   totalisateur_ajouter_roue(plaque, FAIRE_roue_num('0','9'));
   totalisateur_ajouter_roue(plaque, FAIRE_roue_num('0','9'));

   totalisateur_initialiser(plaque);
   for (i = 0; i < nbi; i++)
   {
      totalisateur_avancer(plaque);
      afficher(plaque);
   }
}

static void afficher(totalisateur c)
{
   generateur_aller_au_debut();
   printf("  [ ");
   totalisateur_afficher(c, f);
   printf(" 47 ]\n");
}

static void f(int v)
{
   static int c1;
   static int c2;

   switch (generateur_suivant())
   {
     case 1:
       c1 = v;
       break;

     case 2:
       c2 = v;
       break;

     case 3:
     case 4:
       printf("%c", v);
       break;

     case 5:
       printf("%c %c%c", v, c1, c2);
       break;
   }
}
```

plaque.c

Ce dernier programme n'est pas tout à fait exact : il produit un numéro incorrect, de la forme `[ 000 xx 47 ]` lors de chaque changement de lettre. Le lecteur pourra réfléchir à des extensions possibles permettant de traiter correctement ce cas particulier de gestion de retenue.

```
$ plaque 52020
  [ 001 AA 47 ]
  [ 002 AA 47 ]
  [ 003 AA 47 ]
  [ 004 AA 47 ]
  [ 005 AA 47 ]
        .
        .
        .
  [ 998 AA 47 ]
  [ 999 AA 47 ]
  [ 000 AB 47 ]
  [ 001 AB 47 ]
  [ 002 AB 47 ]
  [ 003 AB 47 ]
        .
        .
        .
  [ 018 CA 47 ]
  [ 019 CA 47 ]
  [ 020 CA 47 ]
$
```

8.6 Une alternative à l'itération

Dans l'exemple précédent la plupart des traitements, comme la gestion des retenues ou la libération des roues, sont effectués par le totalisateur. Il s'agit d'une approche algorithmique classique dans laquelle chaque traitement est exprimé à l'aide d'itérations et de tests.

En particulier, dans le cas d'une structure de liste, chaque traitement de la liste se traduit par un parcours effectué au moyen d'une itération (ou d'un appel récursif) dont le but est de répercuter le traitement sur chaque élément de la liste.

Il est possible de rendre ce traitement plus implicite en descendant le contrôle de l'itération au niveau de chaque élément de la liste. C'est d'ailleurs comme cela que les choses se passent dans un totalisateur réel, construit avec de vraies roues. La solution consiste à chaîner directement les roues entre elles. Chaque roue peut être attachée à une autre roue qui sera appelée sa *roue de retenue*. Cette nouvelle organisation des données est représentée sur la figure 8.8.

La gestion de ce chaînage nécessite l'extension de l'interface et de l'implémentation de la classe `roue_num`, avec l'ajout de deux nouvelles méthodes :

- `roue_num_attacher(roue_num recv, roue_num r)` : attache la roue `r` comme roue de retenue du receveur;

– `roue_num_detacher(roue_num)` : détache la roue de retenue du receveur.

On notera que, encore une fois, il a été possible d'effectuer une modification qui préserve la compatibilité ascendante. En effet, il est toujours possible de définir des roues sans attache, qui conservent le même comportement que dans l'implémentation précédente. La nouvelle interface de la classe `roue_num` est la suivante :

```
═══ roue_num.h (avec gestion d'une retenue) ═══
#ifndef ROUE_NUM_H
#define ROUE_NUM_H

#include "objet.h"

CLASSE(roue_num);

extern roue_num FAIRE_roue_num(int, int);
extern void     DEFAIRE_roue_num(roue_num);
extern void     roue_num_attacher(roue_num, roue_num);
extern void     roue_num_detacher(roue_num);
extern void     roue_num_initialiser(roue_num);
extern int      roue_num_avancer(roue_num);
extern int      roue_num_valeur(roue_num);
extern void     roue_num_afficher(roue_num, void (*)(int));

#endif  /* ROUE_NUM_H */
═══ roue_num.h (avec gestion d'une retenue) ═══
```

Nous verrons plus loin la raison d'être de la méthode `roue_num_afficher`.

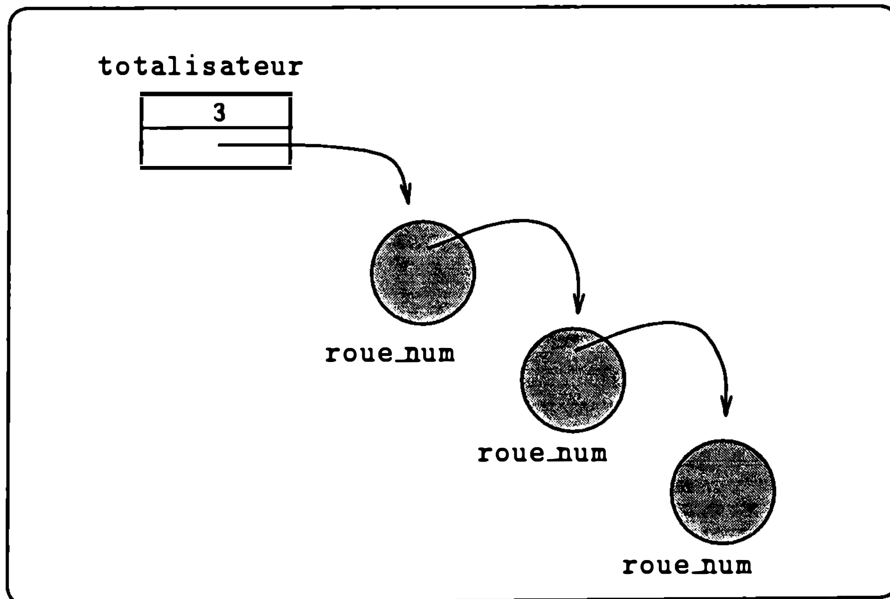


Figure 8.8: Codage d'un totalisateur par chaînage de roues (b)

L'organisation des données a été notablement simplifiée, puisqu'il n'est plus besoin de gérer au niveau du totalisateur des emplacements pour chaîner les roues. On va voir qu'il en est globalement de même avec les traitements. Con-

sidérons tout d'abord les modifications à apporter à l'implémentation de la classe roue_num. Un champ

```
struct roue_num *retenue;
```

est ajouté à la structure struct roue_num pour coder le chaînage de la roue de retenue. Les méthodes roue_num_attacher et roue_num_detacher gèrent ce chaînage.

Le changement le plus important concerne les modifications apportées aux méthodes roue_num_initialiser et roue_num_avancer. Comme précédemment, la méthode d'initialisation roue_num_initialiser ramène le receveur à sa position d'origine, mais elle répercute en plus l'initialisation sur son éventuelle retenue :

```
mise_a_zero(recv);
if (recv->retenue != NULL)
    roue_num_initialiser(recv->retenue);
```

L'initialisation est effectuée par la fonction auxiliaire mise_a_zero dont nous allons voir la raison d'être un peu plus loin. La méthode roue_num_avancer gère directement le traitement de la retenue en envoyant le message avancer à la roue de retenue du receveur (s'il y en a une), à chaque passage par la position d'origine :

```
if (recv->valeur_courante == recv->valeur_maximale)
{
    mise_a_zero(recv);
    if (recv->retenue != NULL)
        roue_num_avancer(recv->retenue);
    return 0;
}
⋮
```

La méthode roue_num_avancer ne peut plus faire directement appel, comme dans les cas précédents, à la méthode roue_num_initialiser pour ramener la roue au début, puisque l'initialisation est répercutée sur toutes les roues attachées. Il a donc été nécessaire de définir une fonction auxiliaire mise_a_zero pour effectuer ce travail.

La méthode de libération DEFAIRE_roue_num est également construite sur ce principe.

roue_num.c (*avec retenue*)

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

#include "classe.h"
#include "roue_num.h"

struct roue_num
{
    int valeur_initiale;
    int valeur_courante;
    int valeur_maximale;
    struct roue_num *retenue;
};
```

```
static void mise_a_zero(roue_num);

roue_num FAIRE_roue_num(int valeur_initiale,
                        int valeur_maximale)
{
    INSTANCIER(roue_num);

    recv->valeur_initiale = valeur_initiale;
    recv->valeur_maximale = valeur_maximale;
    recv->retenue = NULL;
    mise_a_zero(recv);

    return recv;
}

void DEFAIRE_roue_num(roue_num recv)
{
    if (recv->retenue != NULL)
        DEFAIRE_roue_num(recv->retenue);
    free(recv);
}

void roue_num_attacher(roue_num recv, roue_num retenue)
{
    recv->retenue = retenue;
}

void roue_num_detacher(roue_num recv)
{
    recv->retenue = NULL;
}

void roue_num_initialiser(roue_num recv)
{
    mise_a_zero(recv);
    if (recv->retenue != NULL)
        roue_num_initialiser(recv->retenue);
}

int roue_num_avancer(roue_num recv)
{
    assert(recv->valeur_courante <= recv->valeur_maximale);

    if (recv->valeur_courante == recv->valeur_maximale)
    {
        mise_a_zero(recv);
        if (recv->retenue != NULL)
            roue_num_avancer(recv->retenue);
        return 0;
    }
    else
    {
        (recv->valeur_courante)++;
        return 1;
    }
}
```

```
int roue_num_valeur(roue_num recv)
{
    return recv->valeur_courante;
}

void roue_num_afficher(roue_num recv, void (*f)(int))
{
    if (recv->retenue != NULL)
        roue_num_afficher(recv->retenue, f);
    f(recv->valeur_courante);
}

static void mise_a_zero(roue_num recv)
{
    recv->valeur_courante = recv->valeur_initiale;
}
```
roue_num.c (*avec retenue*)

On remarque que le traitement de l'affichage a également été descendu au niveau des roues. Il ne peut en être autrement, le totalisateur n'étant plus en mesure d'effectuer lui-même le parcours de toutes ses roues. L'implémentation de la classe `totalisateur` a été considérablement simplifiée. Chaque itération sur une liste a été remplacée par un envoi de message sur le premier élément de la liste qui se charge de le répercuter ou non sur son successeur. L'interface de la classe demeure inchangée.

totalisateur.c (*avec retenue*)
```
#include <stdio.h>
#include <stdlib.h>

#include "classe.h"
#include "roue_num.h"

#include "totalisateur.h"


struct totalisateur
{
    int      nombre_de_roues;
    roue_num premiere_roue;
};

totalisateur FAIRE_totalisateur()
{
    INSTANCIER(totalisateur);

    recv->nombre_de_roues = 0;
    recv->premiere_roue = NULL;
    return recv;
}

void DEFAIRE_totalisateur(totalisateur recv)
{
    if (recv->premiere_roue != NULL)
        DEFAIRE_roue_num(recv->premiere_roue);
    free(recv);
```

```
}

void totalisateur_ajouter_roue(totalisateur recv, roue_num r)
{
    roue_num_attacher(r, recv->premiere_roue);
    recv->premiere_roue = r;
}

void totalisateur_initialiser(totalisateur recv)
{
    roue_num_initialiser(recv->premiere_roue);
}

void totalisateur_avancer(totalisateur recv)
{
    roue_num_avancer(recv->premiere_roue);
}

int totalisateur_nombre_de_roues(totalisateur recv)
{
    return recv->nombre_de_roues;
}

void totalisateur_afficher(totalisateur recv, void (*f)(int))
{
    roue_num_afficher(recv->premiere_roue, f);
}
```

totalisateur.c (*avec retenue*)

On peut s'interroger sur l'efficacité de cette approche par rapport à la gestion classique d'une itération. Nous avons légèrement modifié le programme `compteur` de manière à ce qu'il ne fasse afficher le totalisateur que deux fois, une première fois avant de lancer les incrémentations, et une seconde après avoir terminé. Nous l'avons compilé avec chacune des deux implémentations, celle avec l'itération explicite effectuée au niveau du totalisateur, et celle avec le chaînage de retenues.

```
$ time compteur_avec_iteration 7 2000000
  [ <0> <0> <0> <0> <0> <0> <0> ]
  [ <2> <0> <0> <0> <0> <0> <0> ]
        6.1 real         5.8 user         0.1 sys
$
$ time compteur_avec_retenue 7 2000000
  [ <0> <0> <0> <0> <0> <0> <0> ]
  [ <2> <0> <0> <0> <0> <0> <0> ]
        5.6 real         5.2 user         0.0 sys
$
```

On constate que les deux programmes s'exécutent sensiblement à la même vitesse, avec un très léger avantage pour la version avec retenues. Un test avec optimisation du code généré affiche la même tendance.

```
$ time compteur_avec_iteration_O 7 2000000
  [ <0> <0> <0> <0> <0> <0> <0> ]
```

```
  [ <2> <0> <0> <0> <0> <0> <0> ]
       2.9 real        2.7 user        0.0 sys
$
$ time compteur_avec_retenue_0 7 2000000
  [ <0> <0> <0> <0> <0> <0> <0> ]
  [ <2> <0> <0> <0> <0> <0> <0> ]
       2.8 real        2.5 user        0.0 sys
$
```

En fait, il serait plus judicieux d'utiliser plusieurs chaînages différents pour relier les roues. Cela permettrait par exemple de traiter dans un ordre différent la transmission des retenues et le balayage des roues pour l'affichage. On peut même étendre ce modèle de façon à permettre de chaîner simultanément plusieurs roues à la suite d'une même roue. Le problème que nous avions laissé en suspens avec l'exemple des plaques minéralogiques peut être résolu de cette façon. La figure 8.9 montre deux chaînages se superposant :

- un chaînage en traits fins implémentant la transmission des retenues;
- un chaînage en traits épais implémentant le contrôle de l'affichage.

La figure 8.10 montre le résultat de la transmission de retenue lors du changement de lettre. La fonction d'affichage est considérablement simplifiée par la présence du chaînage d'affichage. Ce dernier chaînage peut-être utilisé pour le contrôle de l'initialisation.

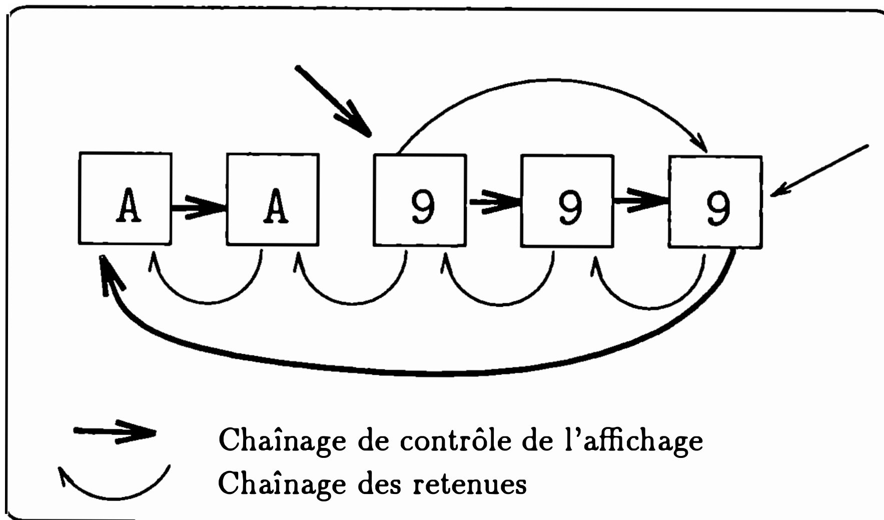


Figure 8.9: Chaînages de contrôle pour la génération de plaques minéralogiques

Cette approche se prête tout à fait à un style de programmation, appelé parfois *programmation visuelle*, visant à construire des programmes au moyen d'outils graphiques interactifs.

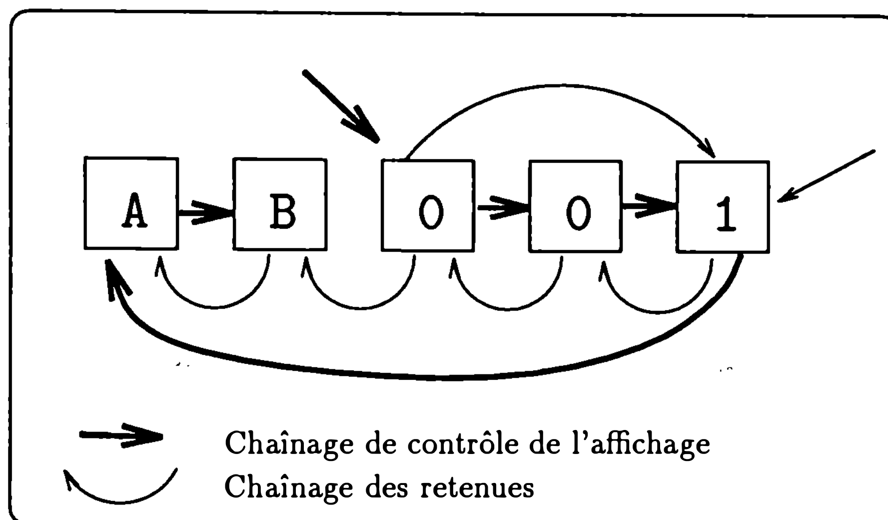


Figure 8.10: Transmission de retenue

8.7 Objets génériques

8.7.1 Modules génériques ou objets génériques

La plupart des classes que nous avons considérées tout au long de ce chapitre décrivent des objets complètement spécifiés. Elles sont facilement réutilisables car totalement protégées. L'anneau de caractères présentée en 8.4.3 soulève cependant un problème. Si l'on a besoin d'un anneau contenant autre chose que des caractères, comme un anneau de chaînes de caractères ou de générateurs d'entiers, il n'est pas possible de réutiliser tel quel le code de la classe `c_anneau`. En effet, celle-ci a été explicitement écrite, au niveau de l'interface comme à celui de l'implémentation, pour gérer des caractères.

Ce problème se généralise à toutes les classes implémentant des collections d'objets : vecteurs, piles, files, listes, etc. La nécessité, dans les langages à typage statique comme C, d'attribuer un type à toute construction, constitue un obstacle réel à la réutilisabilté.

Les techniques utilisées pour résoudre ce problème sont appelées des techniques de **généricité**. Deux stratégies de généricité s'opposent. La première a déjà été présentée en 5.5 avec l'exemple d'une pile. Cette technique consiste à paramétrer la définition du module par un type, puis à instancier une implémentation spécifique pour chaque type nécessaire. Par exemple, dans le cas de la pile, on définit une construction générique `PILE_DECLARER(T, ...)` où `T` est le paramètre définissant le type des éléments de la pile. Si l'on a besoin d'une pile de chaînes de caractères, on écrira simplement `PILE_DECLARER(char *, ...)` pour générer le code spécifique à cette pile[8].

Cette approche est en contradiction avec le principe de masquage de l'implémentation qui est à la base de l'abstraction modulaire. En effet, l'implémentation doit être exportée dans tout programme utilisateur du module de pile, puisque c'est là que se fait l'instanciation de chaque pile typée. On peut expri-

[8]Ce modèle est implémenté dans plusieurs langages de programmation : paquetages génériques en ADA, classes génériques en EIFFEL, *templates* en C++, etc.

mer cela d'une autre manière : il n'est pas possible de compiler un module de pile et d'archiver le programme objet de ce module une fois pour toutes dans une bibliothèque; le code source est nécessaire à chaque instanciation d'une pile typée, et il est nécessaire de recompiler autant d'occurrence du module qu'il y de piles des type différents. Nous dirons, pour résumer cette approche, que c'est le module qui est générique, et non l'objet implémenté.

L'autre approche consiste à utiliser, pour déclarer les éléments de la collection d'objets (pile ou autre), un **type générique** pouvant coder tous les types possibles. En C, la solution consiste à utiliser le type *pointeur générique* `void *` qui, s'il ne peut pas représenter tous les types possibles du langage, peut néanmoins coder un pointeur vers n'importe lequel de ces types. Une pile implémentée selon ce principe sera déclarée comme une pile d'éléments de type `void *`. Avec cette solution, le prototype des méthodes d'empilement et de dépilement est :

```
void pile_empiler(pile, void*);
void *pile_depiler(pile);
```

Les conversions entre un pointeur générique et tout autre pointeur étant implicites, si p est de type `pile`, il est possible d'écrire

```
pile_empiler(p, e)
```

ou

```
e = pile_depiler(p)
```

pour n'importe quel pointeur `e`.

Contrairement au cas précédent, nous dirons dans ce cas que c'est l'objet implémenté qui est générique et non sa description. En résumé, dans le premier cas nous avons défini un *module générique* de pile, et dans le second un module de *pile générique*. Du point de vue du masquage de l'implémentation, seule la seconde solution est vraiment modulaire.

8.7.2 La classe "vecteur générique extensible"

Nous allons illustrer ce propos en modularisant une construction que nous avons déjà programmée à plusieurs reprises : le vecteur extensible. Il constitue un candidat idéal à la généricité car il est susceptible d'être souvent réutilisé.

Il s'agit d'un vecteur de pointeurs `void *`, se rallongeant automatiquement lorsque cela s'avère nécessaire. Il n'y a donc pas besoin de spécifier sa taille, celle-ci étant gérée automatiquement dans l'implémentation de la classe.

Les deux méthodes donnant accès aux éléments du vecteur, `vecteur_lire` et `vecteur_ecrire`, s'utilisent avec en paramètre l'indice de l'élément à lire ou à modifier. Selon l'usage courant en C, l'indiçage commence à partir de zéro. La méthode `vecteur_nombre_elements` retourne la taille courante du vecteur. Enfin, la méthode `vecteur_mode_bavard` permet d'activer ou de désactiver un mode dans lequel chaque rallongement du vecteur est signalé par un message sur la sortie standard erreur.

```
═══════════ vecteur.h ═══════════════════════════════
#ifndef VECTEUR_H
#define VECTEUR_H
```

```
#include "objet.h"

CLASSE(vecteur);

extern vecteur FAIRE_vecteur(void);
extern void DEFAIRE_vecteur(vecteur);
extern void vecteur_ecrire(vecteur, int, void *);
extern void *vecteur_lire(vecteur, int);
extern int  vecteur_nombre_elements(vecteur);
extern void vecteur_mode_bavard(vecteur, int);

#endif  /* VECTEUR_H */
```
vecteur.h

Le contenu du vecteur générique est implémenté au moyen d'un vecteur de pointeurs génériques. L'implémentation utilise quatre variables d'instance :

- `debut`, codant le début du contenu du vecteur;
- `taille`, codant la taille physique du vecteur, c'est-à-dire la taille du bloc mémoire référencé par début;
- `nombre_elements`, codant la taille logique du vecteur, définie par le plus grand indice utilisé;
- `mode_bavard`, valant *vrai* lorsque le mode bavard a été demandé.

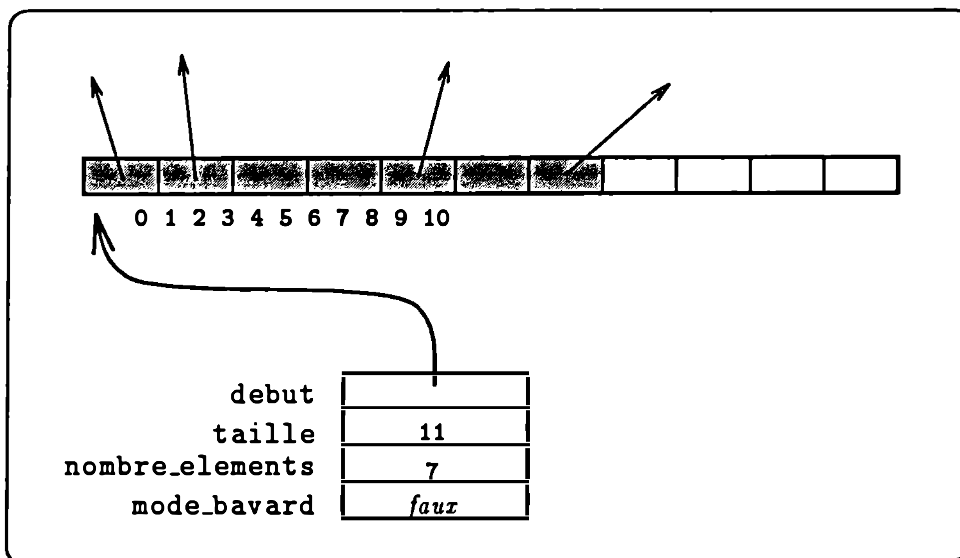


Figure 8.11: Implémentation du vecteur générique extensible

Le rallongement du contenu du vecteur se fait par blocs dont la taille est un multiple de la constante

`NB_ELEM_PAR_BLOC * sizeof(void *)`

Pour les besoins de la démonstration, le nombre d'éléments par blocs a été fixé à 3.

Le vecteur est automatiquement rallongé lors de l'envoi du message `ecrire` avec un indice i provoquant un débordement. Dans ce cas, la nouvelle taille

est le premier multiple de NB_ELEM_PAR_BLOC supérieur à i. Ce calcul est réalisé par l'expression

i + (NB_ELEM_PAR_BLOC - (i % NB_ELEM_PAR_BLOC))

La première allocation est effectuée au moyen de **CALLOC(1)** qui initialise à zéro la zone allouée. Toute nouvelle case allouée est initialisée à NULL. Toute tentative d'accès en lecture à un élément en dehors de la taille logique du vecteur provoque une erreur.

vecteur.c

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

#include "classe.h"
#include "vecteur.h"

#define NB_ELEM_PAR_BLOC    3

struct vecteur
{
    void **debut;
    long taille;
    long nombre_elements;
    int  mode_bavard;
};

vecteur FAIRE_vecteur(void)
{
    INSTANCIER(vecteur);

    recv->debut = calloc(NB_ELEM_PAR_BLOC, sizeof(void*));
    EXIT_IF(recv->debut==NULL, "Erreur allocation dynamique");

    recv->taille = NB_ELEM_PAR_BLOC;
    recv->nombre_elements = 0;
    recv->mode_bavard = 0;
    return recv;
}

void DEFAIRE_vecteur(vecteur recv)
{
    free(recv->debut);
    free(recv);
}

void vecteur_ecrire(vecteur recv, int indice, void *valeur)
{
    void **p;

    if (indice >= recv->taille)
    {
        int n = indice + (NB_ELEM_PAR_BLOC
                          - (indice % NB_ELEM_PAR_BLOC));

        recv->debut = realloc(recv->debut, n * sizeof(void*));
        for (p = recv->debut + recv->taille;
```

```
             p < recv->debut + n; p++)
            *p = NULL;

        if (recv->mode_bavard)
        {
            fprintf(stderr, ">>vecteur: extension de %d a %d\n",
                    recv->taille, n);
        }
        recv->taille = n;
    }
    recv->debut[indice] = valeur;
    if (recv->nombre_elements < indice+1)
        recv->nombre_elements = indice+1;
}

void *vecteur_lire(vecteur recv, int indice)
{
    assert(indice >= 0);
    assert(indice<recv->nombre_elements);
    return recv->debut[indice];
}

int vecteur_nombre_elements(vecteur recv)
{
    return recv->nombre_elements;
}

void vecteur_mode_bavard(vecteur recv, int mode)
{
    recv->mode_bavard = mode;
}
```
vecteur.c

Voici un exemple d'utilisation du vecteur générique. Il s'agit d'une commande instanciant un vecteur et écrivant successivement dans les cases d'indice 2, 7, 0 et 10. La pseudo-fonction TRACE permet d'obtenir une trace de l'exécution de chaque écriture, et le mode bavard est positionné à *vrai* afin d'obtenir celle des rallongements. Le programme se termine par un listage de tous les éléments du vecteur, et une tentative d'accès au delà de sa limite.

test_vecteur.c

```
#include "vecteur.h"

#define TRACE(e) (printf("%s:%d:%s\n", \
                         __FILE__, __LINE__, #e), e)

main()
{
    vecteur v = FAIRE_vecteur();
    int i;
    int n;

    vecteur_mode_bavard(v, 1);

    TRACE(vecteur_ecrire(v, 2, "deux"));
    TRACE(vecteur_ecrire(v, 7, "sept"));
```

```
    TRACE(vecteur_ecrire(v, 0, "zero"));
    TRACE(vecteur_ecrire(v, 10, "dix"));

    n = vecteur_nombre_elements(v);
    for (i=0; i<n; i++)
        printf(" v[%d] : %s\n", i, vecteur_lire(v, i));
    TRACE(vecteur_lire(v, n));
}
```

test_vecteur.c

Cette démonstration est effectuée avec des blocs de rallongement de trois éléments. C'est également la taille initiale du vecteur. La première écriture peut donc être effectuée sans rallongement. Pour pouvoir effectuer la seconde, la taille du vecteur est augmentée, la nouvelle taille étant le plus petit multiple de 3 strictement supérieur à 7, soit l'entier 9. De même, l'affectation de la douzième case du vecteur, la case d'indice 10, nécessite une rallongement de 9 à 12 cases.

Une fois les écritures terminées, la taille courante du vecteur est de 11 cases. Il s'agit ici de la taille logique, la seule connue hors de l'implémentation. La taille physique est par contre de 12 cases. La tentative de lecture de la onzième case provoque un erreur.

```
$ test_vecteur
test_vecteur.c:13:vecteur_ecrire(v, 2, "deux")
test_vecteur.c:14:vecteur_ecrire(v, 7, "sept")
>>vecteur: extension de 3 a 9
test_vecteur.c:15:vecteur_ecrire(v, 0, "zero")
test_vecteur.c:16:vecteur_ecrire(v, 10, "dix")
>>vecteur: extension de 9 a 12
 v[0] : zero
 v[1] : (null)
 v[2] : deux
 v[3] : (null)
 v[4] : (null)
 v[5] : (null)
 v[6] : (null)
 v[7] : sept
 v[8] : (null)
 v[9] : (null)
 v[10] : dix
test_vecteur.c:21:vecteur_lire(v, n)
vecteur.c:64: failed assertion 'indice<recv->nombre_elements'
Abort (core dumped)
$
```

8.7.3 La classe "pile générique"

Voici un second exemple, celui d'une pile générique, implémentée au moyen de la classe `vecteur` décrite précédemment. L'interface de cette classe est très classique: fonctions d'empilement, dépilement, test à *pile vide* et retour de

la valeur du sommet de pile. Il n'y a bien sûr pas de méthode `pile_pleine` puisqu'aucune restriction de taille n'est imposée.

```
═══════ pile.h ═══════
#ifndef PILE_H
#define PILE_H

#include "objet.h"

CLASSE(pile);

extern pile FAIRE_pile(void);
extern void DEFAIRE_pile(pile);
extern void pile_empiler(pile, void*);
extern void *pile_depiler(pile);
extern int  pile_vide(pile);
extern void *pile_sommet(pile);

#endif  /* PILE_H */
═══════ pile.h ═══════
```

L'implémentation de la pile est particulièrement simple. La variable d'instance `contenu`, de type `vecteur`, code les éléments de la pile au moyen d'un vecteur extensible de pointeurs génériques, et l'entier `sommet` est l'indice dans le vecteur de la case suivante du sommet de pile. La pile est vide lorsque `sommet` a pour valeur 0. Une tentative de dépilement, ou de lecture du sommet, sur une pile vide provoque une erreur.

```
═══════ pile.c ═══════
#include <stdlib.h>
#include <assert.h>

#include "classe.h"

#include "pile.h"
#include "vecteur.h"

struct pile
{
    vecteur contenu;
    int sommet;
};

pile FAIRE_pile(void)
{
    INSTANCIER(pile);

    recv->contenu = FAIRE_vecteur();
    recv->sommet = 0;
    return recv;
}

void DEFAIRE_pile(pile recv)
{
    DEFAIRE_vecteur(recv->contenu);
    free(recv);
```

```
}

void pile_empiler(pile recv, void *valeur)
{
    vecteur_ecrire(recv->contenu, recv->sommet, valeur);
    (recv->sommet)++;
}

void *pile_depiler(pile recv)
{
    assert(!pile_vide(recv));
     --(recv->sommet);
    return vecteur_lire (recv->contenu, recv->sommet);
}

void *pile_sommet(pile recv)
{
    assert(!pile_vide(recv));
    return vecteur_lire(recv->contenu, recv->sommet - 1);
}

int pile_vide(pile recv)
{
    return (recv->sommet == 0);
}
```

pile.c

Le programme `test_pile` est un exemple d'utilisation de cette classe. Il ne présente pas de difficulté particulière.

test_pile.c

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

static void empiler(pile, char *);
static void depiler(pile);

main()
{
    pile p = FAIRE_pile();

    empiler(p, "un");
    empiler(p, "deux");
    empiler(p, "trois");
    empiler(p, "quatre");
    empiler(p, "cinq");
    depiler(p);
    depiler(p);
    empiler(p, "six");
    empiler(p, "sept");
    empiler(p, "huit");
    empiler(p, "neuf");
    while (!pile_vide(p))
        depiler(p);

    /* Erreur assertion ''non pile_vide'' */
```

```
    pile_sommet(p);
}

void empiler(pile p, char *s)
{
    printf(" [emp]: %s\n", s);
    pile_empiler(p, s);
}

void depiler(pile p)
{
    printf(" [dep]: %s\n", (char *)pile_depiler(p));
}
```
test_pile.c

Voici pour terminer le résultat de l'exécution de cette commande.

```
$ test_pile
 [emp]: un
 [emp]: deux
 [emp]: trois
 [emp]: quatre
pile_g: extension de 3 a 6
 [emp]: cinq
 [dep]: cinq
 [dep]: quatre
 [emp]: six
 [emp]: sept
 [emp]: huit
 [emp]: neuf
pile_g: extension de 6 a 9
 [dep]: neuf
 [dep]: huit
 [dep]: sept
 [dep]: six
 [dep]: trois
 [dep]: deux
 [dep]: un
pile.c:45: failed assertion '!pile_vide(recv)'
Abort (core dumped)
$
```

8.8 Vers le polymorphisme

Dans ce chapitre, nous avons présenté trois modèles permettant de modulariser une application. Nous avons vu que, sauf pour des cas très particuliers, seuls les deux derniers, c'est-à-dire les modules implémentés par des fichiers et les classes, sont réellement satisfaisants. L'encapsulation par fichier est suffisante tant qu'il n'est pas nécessaire de coder plusieurs occurrences de l'entité implémentée. La généralisation au moyen d'une classe est cependant préférable.

Nous avons vu deux techniques favorisant la réutilisabilité :

- le paramétrage des traitements par des fonctions avec l'approche dirigée par les données;
- la définition de modules génériques.

Ces deux approches peuvent se compléter de façon remarquable.

La faiblesse d'une classe générique est de tout ignorer des objets qu'elle manipule. Elle ne peut pas, par conséquent, effectuer de traitement spécifique sur ces objets, et en particulier l'appel à leur fonction de libération. Par exemple, dans le cas du vecteur présenté en 8.7.2, il est souhaitable que la fonction de libération `DEFAIRE_vecteur` libère automatiquement les objets contenus dans le vecteur avant de détruire le vecteur lui-même. La solution pour permettre cela consiste à conserver, dans chaque vecteur, la référence à la fonction de libération de ses éléments. Celle-ci peut être passée en paramètre au moment de l'instanciation.

Avec ce principe, pour instancier un *vecteur de totalisateurs*, il suffit d'appeler la fonction d'instanciation de `vecteur` avec, en paramètre, la fonction de libération de `totalisateur` :

```
vecteur les_totalisateurs = FAIRE_vecteur(DEFAIRE_totalisateur)
```

Un attribut

```
void (*defaire)(void *)
```

est rajouté dans l'environnement de l'objet vecteur pour conserver ce paramètre. La fonction de libération de la classe `vecteur` est maintenant en mesure d'effectuer correctement son travail :

```
void DEFAIRE_vecteur(vecteur recv)
{
    int i;

    for (i=0; i < recv->nombre_elements; i++)
    {
        void *p = recv->debut[i];

        if (*p != NULL)
            recv->defaire(p);
    }
    free(recv->debut);
    free(recv);
}
```

Cette façon de faire peut être étendue à une liste de comportements : méthode d'initialisation, méthode d'affichage.... Il est alors possible de gérer de manière homogène une collection d'objets, partageant une même interface, mais possédant des implémentations spécifiques. Cette technique est également appelé *polymorphisme*.

Troisième partie

Interface avec le système Unix

Chapitre 9

Les appels système

9.1 Interface avec le noyau UNIX

Le noyau du système UNIX est composé d'un ensemble de fonctions gérant :

- l'organisation des systèmes de fichiers sur disque;
- le fonctionnement des entrées-sorties;
- la création et l'ordonnancement des processus, et les communications entre processus;
- l'implantation des processus en mémoire.

Cet ensemble de fonctions constitue un programme monolithique, résidant en permanence dans la mémoire de l'ordinateur.

Un programme utilisateur s'exécute dans un mode d'adressage logique différent de l'adressage physique. Par exemple, son adresse *zéro* n'est pas l'adresse zéro véritable de la mémoire, mais une adresse exprimée relativement à sa partition utilisateur.

Lors de son utilisation par une instruction machine, cette adresse est convertie en adresse absolue par une composante matérielle du processeur, l'**Unité de Gestion de la Mémoire** ou *MMU* (de l'anglais *Memory Management Unit*). Le *MMU* effectue cette conversion en fonction de tables initialisées par le noyau.

Par conséquent, il n'est pas possible à un processus d'accéder directement aux fonctions du noyau, ni aux données qu'il manipule, celles-ci étant situées dans une partie de la mémoire qui lui est cachée.

Cependant, il existe un ensemble de fonctions d'interface, appelées les **appels système**, permettant de mettre en œuvre depuis un processus utilisateur la plupart des fonctionnalités du noyau.

L'invocation d'un appel système s'effectue au moyen d'un mécanisme d'interruption que nous ne détaillerons pas ici, dont la mise en œuvre est réalisée en langage machine, à l'aide d'une instruction machine spécifique. Il est possible d'invoquer les appels système directement depuis un programme C. En effet, pour chaque appel implémenté dans le noyau, est définie une fonction C du même nom provoquant le déroutement du programme vers cet appel; ces fonctions d'interface gèrent également la transmission des paramètres et

la récupération de la valeur de retour. Il n'est donc pas nécessaire, comme c'est souvent le cas sur d'autres systèmes, d'écrire des programmes en langage machine. La description des fonctions d'interface avec les appels système fait l'objet du chapitre deux du manuel UNIX.

9.2 Récupération des erreurs

En cas d'erreur, tous les appels retournent la valeur −1. La variable globale `errno` reçoit le code de cette erreur. La liste des codes d'erreur, contenue dans le fichier en-tête `errno.h`, est décrite dans la page **INTRO(2)** du manuel UNIX. Par exemple, l'erreur EISDIR est provoquée par une tentative d'écriture sur un répertoire[1]. Voici une portion de code effectuant un appel à la fonction `open` d'initialisation d'une entrée-sortie; si l'appel échoue, la valeur de `errno` permet d'identifier la cause précise de cet échec :

```
fd = open(nom, mode);
if (fd == -1)
    switch (errno)
    {
      /* Tentative d'ecriture sur un repertoire */
      case EISDIR:
        ...
      /* Pas de droit d'acces */
      case EACCES:
        ...
```

À chaque erreur est associé un message standard qu'il est possible de faire afficher au moyen de la fonction de bibliothèque **PERROR(3)**. La syntaxe de cette fonction est

void perror(char *⟨*chaîne*⟩)

Le message d'erreur imprimé est formé de la chaîne passée en paramètre, suivi d'un deux-points et du message d'erreur standard correspondant à la valeur courante de `errno`. Le message est terminé par un *newline*.

Il est possible d'accéder directement au message d'erreur. La liste de tous les messages est rangée dans un vecteur de chaînes appelé `sys_errlist`, déclaré en variable globale. Le message d'erreur associé à la valeur courante de `errno` est donné par l'expression

sys_errlist[errno]

L'exemple suivant illustre ces différentes façons de gérer l'affichage d'un message d'erreur, soit en utilisant directement le vecteur `sys_errlist`, soit au moyen de la fonction `perror`.

errno.c

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

[1] Attention, la variable `errno` n'est pas réinitialisée à chaque appel. Sa valeur n'est significative que lorsqu'un appel ou une fonction de la bibliothèque retourne une erreur.

```
#include <stdlib.h>

extern char *sys_errlist[];

main(int argc, char *argv[])
{
    if (argc != 2)
    {
        fprintf(stderr, "Usage: %s <fichier>\n", argv[0]);
        exit(1);
    }

    printf("- open(\"%s\", O_WRONLY) retourne %d\n",
           argv[1], open(argv[1], O_WRONLY));
    printf("    errno: %d\n", errno);
    printf("    sys_errlist[%d]: \"%s\"\n",
           errno,
           sys_errlist[errno]);
    {
        static char *separateur = ": ";
        char *message = malloc(strlen(argv[0])
                               + strlen(separateur)
                               + strlen(argv[1])
                               + 1);

        sprintf(message, "%s%s%s",
                argv[0], separateur, argv[1]);
        fprintf(stderr, "- perror(\"%s\"):\n", message);
        perror(message);
        free(message);
    }
}
```
errno.c

Voici trois exemples d'exécution provoquant une erreur lors de l'exécution de l'appel open :

```
$ errno /etc
- open("/etc", O_WRONLY) retourne -1
    errno: 21
    sys_errlist[21]: "Is a directory"
- perror("errno: /etc"):
errno: /etc: Is a directory
$
$ errno /etc/passwd
- open("/etc/passwd", O_WRONLY) retourne -1
    errno: 13
    sys_errlist[13]: "Permission denied"
- perror("errno: /etc/passwd"):
errno: /etc/passwd: Permission denied
$
$ errno LotusBleu
- open("LotusBleu", O_WRONLY) retourne -1
    errno: 2
    sys_errlist[2]: "No such file or directory"
- perror("errno: LotusBleu"):
```

```
errno: LotusBleu: No such file or directory
$
```

9.3 Identification des ressources

Les appels système servent essentiellement à accéder aux ressources physiques de la machine et aux ressources du système. Le contrôle d'accès à une ressource est effectué par le noyau en fonction de deux paramètres :

- l'identité du processus qui demande l'accès,
- l'identité de la ressource.

Un numéro unique est attribué à chaque utilisateur par l'administrateur du système à la création de son compte. Ce numéro est l'identificateur de l'utilisateur. D'autre part, l'ensemble de tous les utilisateurs est partitionné en groupes[2]; chaque groupe est identifié par un numéro unique.

Par conséquent, chaque utilisateur défini sur la machine est identifié par deux informations :

- son **identificateur utilisateur**, ou *uid* (abréviation de *user-id*);
- son **identificateur de groupe**, ou *gid* (abréviation de *group-id*), ou sa liste de groupes sous BSD.

Il existe un utilisateur privilégié appelé le **super-utilisateur**, dont le nom est `root` et l'identificateur utilisateur `0`.

Un processus est identifié par un utilisateur propriétaire et par un groupe propriétaire. De même, tout fichier du système de fichiers possède un utilisateur propriétaire et un groupe propriétaire. Dans les deux cas, le groupe propriétaire n'est pas obligatoirement le groupe de l'utilisateur propriétaire.

Toute ressource physique est associée à un fichier appelé **fichier spécial**. Toute ressource physique est par conséquent identifiée par l'utilisateur propriétaire et par le groupe propriétaire du fichier spécial qui la représente. De plus, chaque fichier possède une liste de droits d'accès concernant son utilisateur propriétaire, son groupe propriétaire, et les autres utilisateurs. Lorsqu'un processus essaie d'accéder à une ressource, le noyau UNIX compare l'identité (utilisateur et groupe propriétaire) du processus avec celle de la ressource pour déterminer la liste des droits d'accès à prendre en compte.

Lorsque l'utilisateur propriétaire est le super-utilisateur, le noyau n'effectue pas de contrôle d'accès. Par conséquent, le super-utilisateur a le droit d'accéder à toutes les ressources de la machine et du système. Plus précisément, un processus avec 0 comme *uid* peut exécuter n'importe quel appel système, sans aucune restriction[3].

---

[2]Sous BSD, il est possible de faire appartenir un utilisateur à plusieurs groupes.

[3]On pourra noter que ce concept de *mode privilégié* est purement logiciel, et différent du mode privilégié du processeur permettant par exemple d'exécuter les instructions privilégiées.

Chapitre 10

Entrées-sorties de bas niveau

Ce chapitre est consacré à la programmation des entrées-sorties de bas niveau, c'est-à-dire des fonctions d'entrées-sorties du noyau UNIX. La plupart des fonctionnalités présentées sont supportées par les implémentations de C sous d'autres systèmes (MSDOS, VMS...). Il s'agit alors, en général, d'une simulation des fonctionnalités du noyau UNIX.

10.1 Organisation des entrées-sorties

10.1.1 Pilotes d'entrée-sortie

Les entrées-sorties sont des échanges d'informations entre la mémoire principale de l'ordinateur et les unités périphériques. Ces unités périphériques sont de nature diverse : terminaux, imprimantes, unités de disques ou de bandes magnétiques... Chaque unité physique est pilotée par un composant matériel appelé **contrôleur** qui se présente généralement sous la forme d'une carte électronique. Cette carte réalise l'interface entre le périphérique et le système d'exploitation.

La communication avec un contrôleur d'entrée-sortie s'effectue au moyen d'adresses physiques réservées, dépendant de la machine et du contrôleur. Le protocole de communication est spécifique à chaque contrôleur. Dans un système d'exploitation, les programmes chargés de la communication avec un contrôleur de périphérique sont appelés des pilotes de périphériques (*device drivers* dans la terminologie anglo-saxonne).

Classiquement, les entrées peuvent s'effectuer octet par octet – on parlera alors d'entrées-sorties en **mode caractère**, ou bloc par bloc, encore appelées entrées-sorties en **mode bloc**. Dans ce second cas, une mémoire tampon est nécessaire pour stocker le bloc à lire ou à écrire. Une lecture effectuée au clavier d'une console est typiquement une entrée-sortie en mode caractère; par contre, la lecture d'un fichier sur un disque s'effectue en général en mode bloc.

Sous UNIX, chaque pilote de périphérique est associé à un fichier appelé **fichier spécial**. Ces fichiers sont rangés dans un répertoire spécifique, le répertoire /dev (pour *device*). Il existe deux types de fichiers spéciaux correspondant aux deux modes d'entrées-sorties, appelés fichiers spéciaux blocs et

fichiers spéciaux caractères.

Du point de vue du programmeur, il n'existe pas de différence majeure entre un fichier régulier et un fichier spécial. Le même ensemble d'appels permet de mettre en œuvre des entrées-sorties, indifféremment sur n'importe quel type de fichier. L'exécution d'un processus est par conséquent indépendante des unités physiques ou logiques sur lesquelles ce processus effectue ses entrées-sorties. Ce principe, appelé parfois principe d'entrées-sorties généralisées, a été une des raisons du succès qu'a connu le système UNIX.

L'homogénéité des entrées-sorties sous UNIX est renforcée par l'existence de pilotes simulant des périphériques physiques. Le plus simple d'entre eux est le pilote **/dev/null** qui simule un périphérique d'entrées-sorties rudimentaire dont le comportement est le suivant :

- en lecture : simulation d'un fichier vide
- en écriture : abandon des données écrites.

Les trois exécutions suivantes de la commande **rm** illustrent le comportement de ce pilote en écriture. La suppression est demandée sur un fichier inexistant, ce qui provoque l'émission d'un message d'erreur sur la sortie d'erreur standard. La seconde fois la sortie erreur est redirigée dans le fichier **rm.erreurs**, et la troisième, dans le pilote **/dev/null** ce qui a pour effet de faire disparaître toute trace de ce message.

```
$ rm toto
rm: toto: No such file or directory
$ rm toto 2> rm.erreurs
$ cat rm.erreurs
rm: toto: No such file or directory
$ rm toto 2> /dev/null
$
```

Il existe d'autres pilotes de pseudo-périphériques, par exemple ceux simulant des terminaux, ou encore ceux associés aux fenêtres gérées par un gestionnaire de fenêtres. Voici enfin quelques pilotes classiques qu'on trouve sous UNIX :

- **/dev/console** : pilote de la console opérateur de la machine;
- **/dev/kmem** : pilote accédant à la mémoire physique dans laquelle s'exécute le noyau du système;
- **/dev/mem** : pilote accédant à toute la mémoire physique de la machine;
- **/dev/tty** : pilote du terminal auquel est attaché le processus, c'est-à-dire sur lequel s'effectuent les entrées-sorties standard et erreur.

Le nom réel du pilote **tty** du shell courant peut être obtenu au moyen de la commande **TTY(1)** :

```
$ tty
/dev/ttyp1
$
$ TTY=`tty`
$ echo $TTY
/dev/ttyp1
$
```

10.1.2 Descripteurs de fichier

Toute unité d'entrée-sortie est identifiée par un fichier. Les fichiers sont donc les unités logiques d'entrée-sortie. Ils peuvent être :

- des fichiers **réguliers** : ce sont les fichiers contenant des données quelconques (sources, exécutables, textes...);
- des **répertoires** : ce sont des fichiers contenant une liste de noms de fichiers, avec pour chacun le numéro d'identification de ce fichier dans le système de fichiers;
- des fichiers **spéciaux** blocs ou caractères : il contiennent un couple d'entiers, appelés **major** et **minor**, référençant un pilote d'entrée-sortie;
- des **liens symboliques** : ils contiennent le chemin d'un fichier dont ils sont le synonyme;
- suivant les systèmes : des *sockets*, des *pipes nommés*, des *pseudo-tty*... utilisés pour la communication entre processus.

Les principales opérations en relation avec les entrées-sorties sont

- l'initialisation d'une entrée-sortie, encore appelée **ouverture**, et sa terminaison ou **fermeture**;
- les transferts de données en lecture ou en écriture;
- la modification de la position courante dans le cas d'une unité d'entrée-sortie permettant l'accès direct;
- la demande d'informations concernant l'état d'une entrée-sortie en cours;
- la duplication d'une entrée-sortie en cours.

Une **entrée-sortie en cours** est composée des informations suivantes :

- un mode : lecture et/ou écriture;
- éventuellement un index de position courante;
- la référence à un bloc de description du fichier sur lequel s'effectue l'entrée-sortie.

Ces paramètres sont initialisés lors de l'ouverture de l'entrée-sortie. Le bloc de description d'un fichier identifie la liste des fonctions d'entrée-sortie spécifiques à l'unité physique associée. Dans le cas d'un fichier régulier ou d'un répertoire, il contient également un chaînage des blocs de données sur disque.

Une entrée-sortie en cours est identifiée par un entier positif ou nul appelé **descripteur de fichier** (*file descriptor* dans la terminologie anglo-saxonne). Les descripteurs de fichier constituent l'information de plus bas niveau manipulée lors de la programmation des entrées-sorties. Tout appel système effectuant une opération sur une entrée-sortie en cours reçoit comme paramètre principal le descripteur de fichier de cette entrée-sortie.

Les trois entrées-sorties de bas niveau associées à l'entrée standard, la sortie standard et la sortie erreur standard ont respectivement pour descripteur de fichier les entiers 0, 1 et 2.

La figure 10.1 montre un exemple d'environnement d'entrée-sortie d'un processus en cours d'exécution. Les descripteurs 0, 1 et 2 sont associés aux entrées-sorties standard reliées au pilote du terminal (*driver tty*); le descripteur **3** est relié à une entrée-sortie ouverte sur un fichier régulier, en mode *lecture/écriture*. C'est par exemple l'environnement d'entrée-sortie d'un programme ayant effectué une seule initialisation d'entrée-sortie de la forme :

```
int fd = open(donnees_temporaires, O_RDWR|O_CREAT, 0644);
```

On se reportera à la section suivante pour plus de détails sur l'utilisation de l'appel `open`. Sur la figure 10.1, un pilote est représenté par un ensemble de fonctions. Par exemple, le pilote `/dev/tty` est composé des fonctions `read_tty`, `write_tty`... Ce modèle est très proche de la réalité. Un pilote est un programme regroupant les fonctions d'entrées-sorties spécifiques au contrôleur dont il a la charge. Le noyau a pour rôle de relier la fonction d'entrée-sortie générale utilisée par le programmeur, par exemple `read`, avec la fonction spécifique du pilote de périphérique concerné, par exemple `read_tty`; il utilise pour cela le descripteur de fichier référençant l'entrée-sortie en cours.

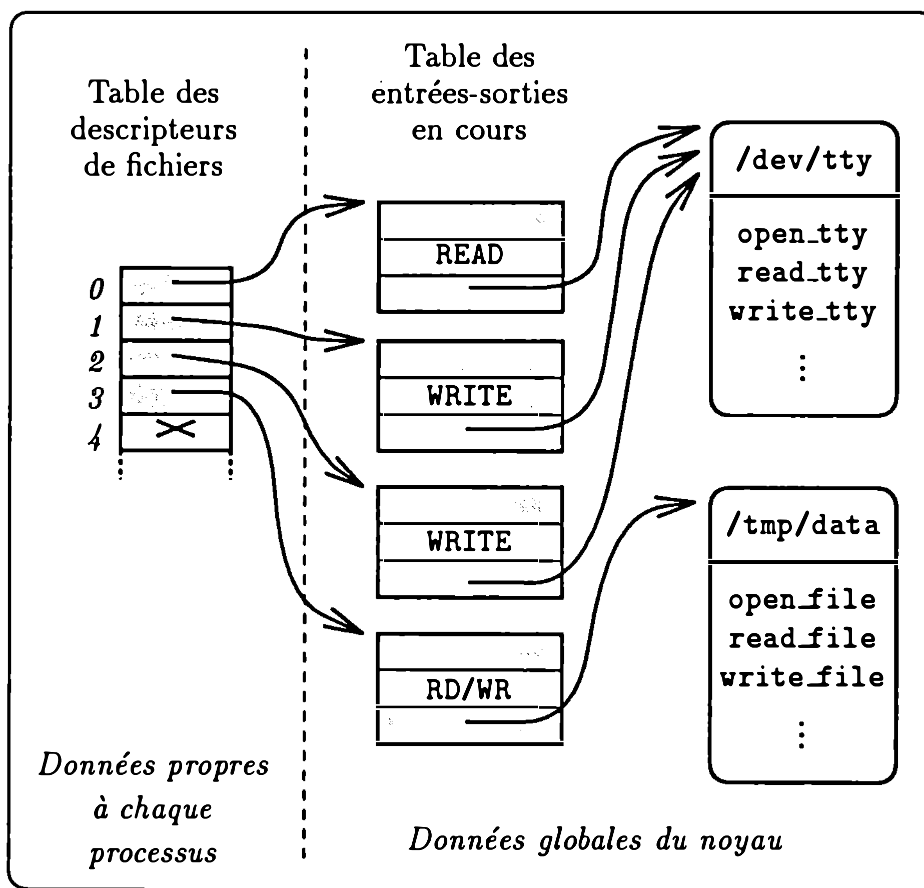


Figure 10.1: Exemple d'environnement d'entrées-sorties

10.2 Mécanismes de base

10.2.1 Ouverture et fermeture

L'initialisation d'une entrée-sortie sur un fichier peut s'effectuer au moyen de deux appels : **OPEN(2)** et **CREAT(2)**.

Initialement, ces deux appels étaient complémentaires, le premier permettant d'accéder à un fichier déjà existant, et le second de créer un nouveau fichier

ou de réinitialiser un fichier déjà existant. L'appel open ayant été étendu, il contient maintenant l'appel creat.

L'appel open

<fcntl.h>

L'appel système open initialise une entrée-sortie et associe cette entrée-sortie à un descripteur de fichier. Il s'utilise avec trois paramètres, le dernier étant optionnel :

int open(char *⟨*chemin*⟩, int ⟨*mode*⟩[, int ⟨*accès*⟩])

Le premier paramètre est le chemin du fichier à ouvrir. Si ce chemin est de l'une des deux formes

$/\langle rep_1\rangle/\langle rep_2\rangle/\cdots/\langle rep_k\rangle/\langle fichier\rangle$
$\langle rep_1\rangle/\langle rep_2\rangle/\cdots/\langle rep_k\rangle/\langle fichier\rangle$

tous les répertoires $\langle rep_1\rangle$ à $\langle rep_k\rangle$ doivent être traversables par l'utilisateur du processus (voir 11.1.2).

Le mode d'accès peut prendre trois valeurs :

- O_RDONLY : lecture seulement;
- O_WRONLY : écriture seulement;
- O_RDWR : lecture et écriture.

Ces constantes sont définies dans le fichier en-tête <fcntl.h>. Sur les anciennes implémentations ne disposant pas de ce fichier, on utilisera respectivement les valeurs 0, 1, et 2.

Si l'ouverture s'effectue avec succès, l'appel retourne le descripteur de fichier identifiant l'entrée-sortie ainsi initialisée. Dans le cas contraire, il retourne -1. Cela implique que tous les descripteurs de fichier sont de valeur positive ou nulle. Par exemple, l'instruction suivante

```
fd = open(argv[1], O_RDONLY);
```

initialise une entrée-sortie en lecture sur le fichier dont le nom est contenue dans la chaîne de caractères argv[1]. L'ouverture peut échouer pour plusieurs raisons : le fichier n'existe pas, il n'est pas accessible en lecture, le chemin d'accès n'est pas traversable, l'utilisateur a déjà ouvert le nombre maximum de fichiers auquel il a droit, etc.

Une ouverture en écriture, c'est-à-dire dans le mode O_WRONLY ou le mode O_RDWR, peut être paramétrée au moyen des options suivantes :

- O_APPEND : toutes les écritures sont effectuées à partir de la fin du fichier, quelle que soit la position courante dans ce fichier;
- O_TRUNC : si le fichier existe déjà il est ramené à une taille nulle.

Les paramètres suivants peuvent s'appliquer à tous les modes d'ouverture :

- O_CREAT : si le chemin ne correspond à aucun fichier existant, le fichier est créé (dans ce cas l'appel open s'utilise avec le troisième paramètre optionnel décrivant les droits d'accès du fichier à créer);
- O_EXCL : provoque une erreur si l'option de création est demandée et si le fichier existe déjà;

- O_NDELAY : dans le cas où l'ouverture nécessite une attente, l'appel open ne se bloque pas et retourne immédiatement une erreur (utile par exemple sous SYSTEM V lors de l'utilisation de *pipes* nommés).

Les options sont définies comme des masques booléens et sont combinables au moyen de l'opérateur booléen |. Par exemple, l'appel

open(nom, O_RDWR|O_CREAT, 0644)

ouvre le fichier nom en mode lecture-écriture, en le créant dans le cas où il n'existe pas. Le troisième paramètre de l'appel open est nécessaire chaque fois qu'un mode d'accès en écriture est utilisé avec l'option O_CREAT. La forme utilisé ici est spécifique au système UNIX : un nombre octal de trois chiffres[1] codant les droits d'accès en lecture-écriture-exécution pour le propriétaire du fichier, les utilisateurs appartenant au même groupe, et les autres utilisateurs. L'accès classique pour un fichier de données est 0644, c'est-à-dire lisible par tout le monde et modifiable par son propriétaire seulement (pour plus de détails sur les modes d'accès, on se reportera à la section 12.1.1). Voici un exemple illustrant ce fonctionnement.

```
═══════ ouvrir.c ═══════════════════════
#include <stdio.h>
#include <fcntl.h>

static void usage(char *);
static int ouvrir(char *);

main(int argc, char *argv[])
{
    if (argc == 2)
        printf("ouvrir renvoie %d\n", ouvrir(argv[1]));
    else
        usage(argv[0]);
}

static int ouvrir(char *nom)
{
    return open(nom, O_RDWR|O_CREAT, 0644);
}

static void usage(char *s)
{
    fprintf(stderr, "Usage: %s fichier\n", s);
    exit(1);
}
═══════════════════════ ouvrir.c ═══════
```

On vérifiera sur l'exemple de session suivant que la fonction ouvrir crée le fichier data s'il n'existe pas, et l'ouvre sans en modifier le contenu s'il existe déjà.

```
$ ls -l data
data not found
```

[1]En réalité, il est possible de donner un quatrième chiffre codant les bits *set-uid*, *set-gid* et *sticky*.

```
$ ouvrir data
ouvrir renvoie 3
$ ls -l data
-rw-r--r--  1 achille        0 Jul 10 15:43 data
$
$ cat > data
Ceci est le contenu du fichier data.
C-d
$ ls -l data
-rw-r--r--  1 achille       37 Jul 10 15:46 data
$ ouvrir data
ouvrir renvoie 3
$ ls -l data
-rw-r--r--  1 achille       37 Jul 10 15:46 data
$
$ ouvrir /data
ouvrir renvoie -1
$
```

Nous traiterons un exemple utilisant l'option O_APPEND en 10.2.3 (page 346), et un autre avec l'option O_NDELAY en 10.3.3 (page 352).

L'appel creat

La création d'un nouveau fichier peut également être effectuée au moyen de l'appel `creat`. Cet appel s'emploie avec deux paramètres :

`int creat(char *⟨chemin⟩, int ⟨accès⟩)`

Lorsque le chemin correspond à un fichier déjà présent, celui-ci est ramené à une taille nulle; dans le cas contraire, un fichier vide est créé. On remarque que dans les deux cas, si l'appel s'exécute normalement le résultat est un fichier vide. L'appel `creat` retourne un descripteur de fichier, ouvert en écriture sur le fichier créé ou initialisé. L'appel `creat` est équivalent à

`open(⟨chemin⟩, O_WRONLY|O_CREAT|O_TRUNC, ⟨accès⟩)`

c'est-à-dire une ouverture

- en mode écriture : mode O_WRONLY,
- avec création si le fichier n'existe pas : option O_CREAT,
- et initialisation sinon : option O_TRUNC.

La raison d'être de cet appel est historique. En effet, dans le premières implémentations d'UNIX, l'appel open ne permettait pas de spécifier des options de mode, ni par conséquent de créer un nouveau fichier ou de réinitialiser un fichier existant. Il existe encore une grande quantité de code écrit avec l'appel `creat` (voir également 11.2.2).

L'appel close

L'appel `close` permet de terminer une entrée-sortie en cours, libérant ainsi les tampons et le descripteur qu'elle utilise. Il reçoit le descripteur de l'entrée-sortie à fermer en paramètre :

```
int close(int ⟨descripteur⟩);
```

Il est préférable de fermer une entrée-sortie dès qu'elle n'est plus utilisée.

L'appel pipe

Le mécanisme du tube UNIX (ou *pipe*) permet d'établir une communication entre deux processus. La version de base est celle du tube anonyme, principalement utilisable par des processus frères, c'est-à-dire créés par un même processus parent. Nous reviendrons sur sa mise en œuvre dans le chapitre 13 consacré aux communications entre processus; nous allons simplement en décrire ici les principes généraux.

L'image d'un tube transportant un flot de caractères illustre assez bien le fonctionnement du tube. Les caractères entrent par une extrémité du tube, et sortent par l'autre. Plus précisément, un tube est une ressource organisée en file, disposant d'un mécanisme de synchronisation de type producteur/consommateur.

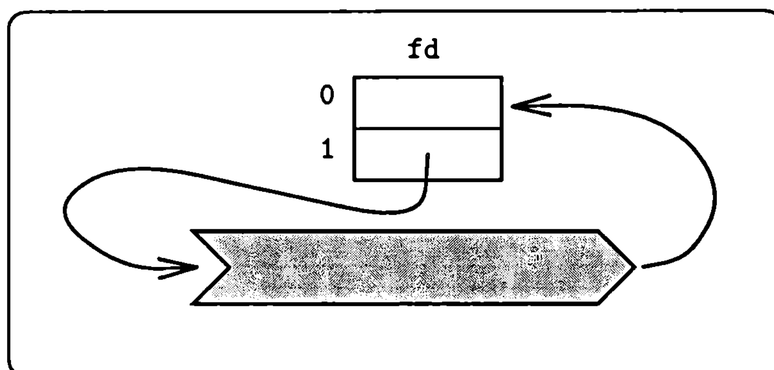


Figure 10.2: Le tube (*pipe*) anonyme.

A un tube est associé de façon naturelle un couple de descripteurs d'entrée-sortie, l'un ouvert en lecture et l'autre en écriture (voir le schéma de la figure 10.2). Ce couple de descripteurs est alloué par l'appel **PIPE(2)** et retourné dans un vecteur de deux entiers passé en paramètre. Une fois le tube créé, il est possible de lire sur le descripteur `fd[0]` et d'écrire sur `fd[1]`.

Le tube étant soumis à un mécanisme de type producteur/consommateur, une requête de lecture est gelée si le tampon du tube est vide; il en est de même pour une écriture lorsque le tampon est plein. Deux processus différents peuvent par conséquent communiquer au moyen d'un tube, en se synchronisant sur les données transmises. On trouvera un premier exemple d'utilisation de tube dans ce chapitre en 10.2.2.

Allocation des descripteurs

Nous avons déjà mentionné le fait que sous UNIX trois descripteurs sont prédéfinis lors du lancement d'un processus, définissant ses entrées et sorties standard. Ces trois descripteurs sont les descripteurs numéro 0, 1 et 2. Les descripteurs

sont des entiers positifs ou nuls, attribués par ordre croissant à chaque initialisation d'une entrée-sortie.

Le nombre total de descripteurs, qui est le nombre maximum de fichiers que peut ouvrir simultanément un processus est défini à la compilation du noyau du système. Il correspond à la pseudo-constante NOFILE définie dans le fichier **<sys/param.h>** (ce fichier contient les différents paramètres du système).

L'exemple suivant est exécuté sur une station *Sun* tournant sous système *OS/4*. Dans cette configuration, le nombre maximum de fichiers ouvrables simultanément par un processus est de 64; les descripteurs disponibles sont donc compris entre 3 et 63, les trois premiers étant déjà alloués.

```
maxfd.c
#include <stdio.h>
#include <fcntl.h>

main()
{
    static int nb_par_ligne = 0;

    printf(" - Descripteurs de fichier disponibles:\n");
    for (;;)
    {
        /* Ouverture en lecture du repertoire courant */
        int fd = open(".", O_RDONLY);

        if (fd == -1)
            break;
        printf("%3d ", fd);
        if (++nb_par_ligne >= 9)
        {
            /* Pour la mise en page */
            nb_par_ligne = 0;
            printf("\n");
        }
    }
    if (nb_par_ligne > 0)
    printf("\n");
}
maxfd.c
```

La commande **GREP(1)** recherche un motif dans une liste de fichiers. Ici, nous l'utilisons pour afficher la valeur de NOFILE.

```
$ grep NOFILE /usr/include/sys/param.h
#define NOFILE  64        /* max open files per process */
$
$ maxfd
- Descripteurs de fichier disponibles:
  3   4   5   6   7   8   9  10  11
 12  13  14  15  16  17  18  19  20
 21  22  23  24  25  26  27  28  29
 30  31  32  33  34  35  36  37  38
 39  40  41  42  43  44  45  46  47
 48  49  50  51  52  53  54  55  56
```

```
 57  58  59  60  61  62  63
$
```

Si un descripteur est libéré au moyen de l'appel système `close`, il pourra être réalloué lors d'une future ouverture. L'algorithme d'allocation d'un descripteur recherche le plus petit numéro disponible.

Il est possible de fermer une entrée-sortie prédéfinie. Dans ce cas, une nouvelle entrée-sortie peut être associée à son descripteur. L'exemple suivant enchaîne diverses ouvertures et fermetures de fichiers en affichant la valeur des différents descripteurs alloués par `open`.

```
ouverture.c
#include <stdio.h>
#include <fcntl.h>

main()
{
    printf("%d  ", open("..", O_RDONLY));
    printf("%d  ", open("..", O_RDONLY));
    printf("%d  ", open("..", O_RDONLY));
    close(3);
    close(5);
    printf("%d  ", open("..", O_RDONLY));
    close(0);
    printf("%d  ", open("..", O_RDONLY));
    printf("\n");
}
ouverture.c
```

À chaque appel à `open`, c'est toujours le plus petit descripteur libre qui est alloué :

```
$ ouverture
3  4  5  3  0
$
```

Les trois premiers appels à `open` allouent successivement les descripteurs 3, 4 et 5. Les suivants allouent le plus petit descripteur libre au moment de l'appel, quel que soit l'ordre dans lequel ils ont été libérés. On remarque qu'il est possible de réallouer les descripteurs prédéfinis. Ce mécanisme de redirection sera développé en 10.3.1.

10.2.2 Lecture et écriture

Les appels read et write

Les deux appels **READ(2)** et **WRITE(2)** permettent d'effectuer respectivement des lectures et des écritures sur une entrée-sortie correctement initialisée. Une lecture n'est possible que si l'entrée-sortie a été ouverte en lecture (modes `O_RDONLY` ou `O_RDWR`). De même, une écriture n'est possible que si l'entrée-sortie a été ouverte en écriture (modes `O_WRONLY` ou `O_RDWR`).

Les opérations de lecture et d'écriture sont des opérations symétriques. Une

lecture est un transfert de données d'une unité d'entrée-sortie vers la mémoire du processus; une écriture est un transfert de données de la mémoire vers une unité d'entrée-sortie.

Les appels `read` et `write` s'emploient avec trois arguments :

```
int read(int ⟨descripteur⟩, char *⟨buffer⟩, int ⟨nb-octets⟩)
int write(int ⟨descripteur⟩, char *⟨buffer⟩, int ⟨nb-octets⟩)
```

Ils retournent le nombre de caractères effectivement transmis. Ce nombre peut être inférieur ou égal au nombre de caractères à transmettre. Par exemple, si `buffer` est un vecteur de 512 caractères, l'exécution de

```
read(0, buffer, sizeof buffer)
```

demande le transfert de 512 caractères depuis l'unité associée à 0, vers la zone mémoire d'adresse *buffer*.

Le nombre n de caractères réellement lus peut être inférieur au nombre de caractères demandé. Par exemple, si l'entrée standard n'a pas été redirigée, la lecture s'effectue au clavier et le nombre de caractères lus est généralement le nombre de caractères tapés jusqu'à un retour chariot. Une valeur de retour de `read` égale à zéro signifie **fin de fichier**. Le programme suivant permet de tester ce comportement :

```
tread.c
#include <stdlib.h>
#include "pluriel.h"

#define TBUF    512

main(int argc, char *argv[])
{
    char buffer[TBUF];
    int nb_demandes = argc > 1 ? atoi(argv[1]) : 1;
    int nb_lus;

    if (nb_demandes > TBUF)
        nb_demandes = TBUF;
    nb_lus = read(0, buffer, nb_demandes);
    printf("-> %d demande%s, %d lu%s\n",
           nb_demandes, PLURIEL(nb_demandes),
           nb_lus, PLURIEL(nb_lus));
}
tread.c
```

Sur l'exemple suivant, la première lecture retourne les quatre caractères `a`, `b`, `c` et ↵. La seconde lit les sept premiers des treize caractères tapés. Les six suivants sont lus et interprétés par le shell. Les troisième et quatrième lectures lisent respectivement un et zéro caractère, correspondant à la frappe de RETURN et de `C-d` en début de ligne.

```
$ tread 7
abc
-> 7 demandes, 4 lus
$ tread 7
abcdefghijkl
-> 7 demandes, 7 lus
```

```
$ hijkl
hijkl: command not found
$ tread 7
C-m
-> 7 demandes, 1 lu
$ tread 7
C-d -> 7 demandes, 0 lu
$
```

Une valeur de retour de `write` inférieure au paramètre ⟨*nb-octets*⟩ signifie qu'il n'y a plus de place sur l'unité physique associée à l'entrée-sortie. Une valeur de retour de `read` ou de `write` égale à -1 correspond à la détection d'une erreur d'entrée-sortie. Il s'agit en général d'une erreur liée au matériel (bloc erroné sur le disque, panne de contrôleur...) face à laquelle le programme ne peut rien, sinon recommencer l'opération d'entrée-sortie un certain nombre de fois avant d'abandonner le traitement.

Les entrées-sorties s'exécutant en mode bloc, notamment celles sur des fichiers réguliers et des répertoires, possèdent un **index de position courante.** Il s'agit de la position à laquelle s'effectuera la prochaine opération de lecture ou d'écriture. Par défaut, l'index de position est initialisé sur le premier caractère du fichier. Une lecture (resp. une écriture) incrémente l'index du nombre de caractères effectivement lus (resp. écrits). Il existe d'autres entrées-sorties en mode bloc, comme les entrées-sorties sur les unités de mémoire `mem` et `kmem`.

Voici maintenant une fonction permettant d'écrire une chaîne sur un descripteur ouvert en écriture :

ecrire.c

```
#include <string.h>
#include "ecrire.h"

int ecrire(int fd, char *s)
{
    return write(fd, s, strlen(s));
}
```

ecrire.c

Elle permet d'écrire

```
ecrire(fd, "Destruction confirmee (O/N) : ")
```

à la place de

```
write(fd, "Destruction confirmee (O/N) : ", 30)
```

Plus généralement, on appliquera la règle :

Lors d'une entrée-sortie de bas niveau, par `read` *ou* `write`*, on n'écrit pas explicitement le nombre de caractères à transférer (excepté, éventuellement, s'il s'agit d'un seul caractère). Lorsqu'il est nécessaire d'exprimer la taille du tampon, on utilise l'opérateur* `sizeof`*.*

Recopie de données

La recopie de données s'effectue en enchaînant des opérations de lecture et d'écriture. Le cas le plus simple consiste à recopier l'entrée standard sur la sortie standard.

```
recopie.c
#define TBUF    1024

main(int argc, char *argv)
{
    char buffer[TBUF];
    int nblus;

    for (;;)
    {
        nblus = read(0, buffer, sizeof buffer);
        if (nblus <= 0)
        {
            if (nblus == -1)
                perror(argv[0]);
            break;
        }
        if (write(1, buffer, nblus) < nblus)
        {
            perror(argv[0]);
            break;
        }
    }
}
recopie.c
```

Les caractères lus sur 0 sont recopiés sur l'unité associée à 1. À chaque itération, ce sont les *nblus* premiers caractères du vecteur `buffer`. La sortie de la boucle de lecture s'effectue lorsque l'appel **read** retourne une valeur négative ou nulle. Le fichier source `recopie.c` peut être listé en tapant la ligne de commande UNIX

```
recopie < recopie.c
```

Voici maintenant une version de recopie de données un peu plus sophistiquée, dans laquelle la lecture s'effectue sur un fichier dont le nom est passé en paramètre. Il suffit pour cela de rajouter un mécanisme d'ouverture et de fermeture au programme précédent.

```
lister.c
#include <stddef.h>
#include <fcntl.h>
#include "lister.h"
#include "concatener.h"

#define TBUF 1024

extern char *argv0;

int lister(char *nom)
```

```
{
    int fd = open(nom, O_RDONLY);
    int code_retour = 1;

    if (fd == -1)
    {
        perror(concatener(argv0,":", nom, NULL));
        return 0;
    }
    for (;;)
    {
        char buffer[TBUF];
        int nblus = read(fd, buffer, sizeof buffer);

        if (nblus == -1)
        {
            code_retour = 0;
            break;
        }
        if (nblus == 0)
            break;
        if (write(1, buffer, nblus) < nblus)
        {
            code_retour = 0;
            break;
        }
    }
    close(fd);
    return code_retour;
}
```
lister.c

Les opérations de lecture et écriture défilent le fichier en séquentiel, à partir de la position courante. Si une entrée-sortie a été initialisée en mode lecture-écriture, il est possible d'alterner dessus des opérations de lecture et d'écriture. Chaque appel à `read` ou `write` incrémente l'index de position courante du nombre de caractères lus ou écrits. Le fichier `readwrite.c` illustre ce comportement.

readwrite.c
```
#include <stddef.h>
#include <stdio.h>
#include <fcntl.h>

#include "concatener.h"
#include "ecrire.h"
#include "lister.h"

char *argv0;

static void verifier_fd(int, char *, char *);
static void defiler_jusqu_a(int, char);

main(int argc, char *argv[])
{
    char *nom = ((argc == 2) ? argv[1]
```

```
                    : concatener(argv[0], ".test", NULL));
    int fd;

    argv0 = argv[0];

    printf("- Creation du fichier %s\n", nom);
    fd = open(nom, O_WRONLY|O_CREAT|O_TRUNC, 0644);
    verifier_fd(fd, argv[0], nom);
    ecrire(fd, "abcdefghijklmnopqrstuvwxyz\n");
    close(fd);

    printf("- Listage du fichier %s\n", nom);
    if (lister(nom) == 0)
        exit(1);

    printf("- Modification du fichier %s\n", nom);
    fd = open(nom, O_RDWR);
    verifier_fd(fd, argv[0], nom);
    ecrire(fd, "<1>");
    defiler_jusqu_a(fd, 'f');
    ecrire(fd, "<2>");
    defiler_jusqu_a(fd, 'y');
    ecrire(fd, "<3>");
    close(fd);

    printf("- Listage apres modification\n");
    lister(nom);
    printf("\n");
}

/* Lecture jusqu'a un caractere donne ou EOF */
static void defiler_jusqu_a(int fd, char c)
{
    for (;;)
    {
        char caractere_lu;

        if (read(fd, &caractere_lu, sizeof caractere_lu) <= 0
            || caractere_lu == c)
            return;
    }
}

static void verifier_fd(int fd, char *cmd, char *arg)
{
    if (fd == -1)
    {
        perror(concatener(cmd, ":", arg, NULL));
        exit(1);
    }
}
```

readwrite.c

Il effectue les opérations suivantes :

- création d'un fichier avec les 26 lettres de l'alphabet et le caractère **\n**;
- listage de ce fichier;

– ouverture du fichier en mode lecture-écriture, alternance de lectures et d'écritures, et fermeture du fichier;
– listage du fichier modifié.

La fonction `defiler_jusqu_a` itère la lecture d'un caractère jusqu'à la lecture d'un caractère donné ou jusqu'à la fin du fichier.

```
$ readwrite FICHIER_TEST
- Creation du fichier FICHIER_TEST
- Listage du fichier FICHIER_TEST
abcdefghijklmnopqrstuvwxyz
- Modification du fichier FICHIER_TEST
- Listage apres modification
<1>def<2>jklmnopqrstuvwxy<3>
$
```

Comme on peut le voir sur cet exemple, chaque nouvelle opération de lecture (resp. d'écriture) débute à la position courante résultant de l'écriture (resp. la lecture) précédente.

Entrées-sorties dans un tube

Si `fd` est un vecteur de deux entiers, l'exécution de

`pipe(fd)`

initialise deux entrées-sorties sur un tube, et place dans `fd[0]` (resp. `fd[1]`) le descripteur de lecture (resp. d'écriture) dans le tube (voir figure 10.2 page 336). Voici un exemple illustrant le fonctionnement des tubes, et enchaînant les opérations suivantes :

– initialisation d'un tube;
– écriture des caractères `abcd` dans le tube : le tube contient la suite de caractères `abcd`;
– lecture d'un caractère : le tube contient la suite de caractères `bcd`;
– écriture de `efg` : le tube contient la suite de caractères `bcdefg`;
– lecture de 1024 caractères : le tube est vide;
– fermeture de l'entrée-sortie ouverte en écriture sur le tube;
– lecture d'un caractère : `read` retourne zéro.

test_tube.c

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

#define CHAINE1  "abcd"
#define CHAINE2  "efg"

static void lire_dans_le_tube(int, int);
static void ecrire_dans_le_tube(int, char *);

main(int argc, char *argv[])
{
    int  fd[2];
```

```
    if (pipe(fd)  == -1)
    {
        perror(argv[0]);
        exit(1);
    }

    ecrire_dans_le_tube(fd[1], CHAINE1);
    lire_dans_le_tube(fd[0], 1);
    ecrire_dans_le_tube(fd[1], CHAINE2);
    lire_dans_le_tube(fd[0], 1024);
    close(fd[1]);
    lire_dans_le_tube(fd[0], 1);
}

static void lire_dans_le_tube(int fd, int n)
{
    char *buffer = malloc(n+1);
    int nb_lus = read(fd, buffer, n);

    buffer[nb_lus] = '\0';
    printf("< lu dans le tube: \"%s\"\n", buffer);
    free(buffer);
}

static void ecrire_dans_le_tube(int fd, char *s)
{
    ecrire(fd, s);
    printf("> ecrit dans le tube: \"%s\"\n", s);
}
```

test_tube.c

Lorsque le nombre de caractères présents dans le tube est inférieur au nombre de caractères demandés, l'appel `read` lit tous les caractères présents et retourne le nombre de caractères effectivement lus.

```
$ test_tube
> ecrit dans le tube: "abcd"
< lu dans le tube: "a"
> ecrit dans le tube: "efg"
< lu dans le tube: "bcdefg"
< lu dans le tube: ""
$
```

Lorsque le tube est vide, l'appel `read` est bloquant s'il reste des entrées-sorties ouvertes en écriture, et retourne zéro sinon. Si, dans le programme précédent, on retire l'avant-dernière instruction de la fonction `main`

```
close(fd[1]);
```

l'exécution du programme se bloque après la quatrième écriture :

```
$ test_tube_sans_close
> ecrit dans le tube: "abcd"
< lu dans le tube: "a"
> ecrit dans le tube: "efg"
< lu dans le tube: "bcdefg"
```

```
C-c
$
```

On est obligé de forcer sa terminaison, par exemple au moyen du caractère de terminaison C-c.

10.2.3 Positionnement en accès direct

Sous UNIX, tous les fichiers de données, c'est-à-dire les fichiers réguliers et les répertoires, sont utilisables en accès direct. Nous avons vu que, par défaut, les opérations de lecture et d'écriture défilent séquentiellement le fichier en incrémentant un index de position courante. Il est très simple de modifier cet index et de fixer ainsi l'emplacement à partir duquel s'effectuera la prochaine lecture ou écriture. Ce mécanisme est géré par l'appel **LSEEK(2)** qui s'emploie avec trois paramètres :

`int lseek(int ⟨descripteur⟩, long ⟨décalage⟩, int ⟨origine⟩)`

Le deuxième paramètre indique le décalage à appliquer à l'index de position, et le troisième l'origine de ce décalage :

- 0 : décalage à partir du début du fichier;
- 1 : décalage à partir de la position courante;
- 2 : décalage à partir de la fin du fichier.

Le décalage peut être une quantité négative. La fonction `lseek` retourne la valeur de la nouvelle position courante, une fois le décalage effectué. Cette position est exprimée en octets à partir du début du fichier. On peut donc connaître la position courante en effectuant un décalage de 0 octet à partir de la position courante, ce qui s'écrit :

`lseek(fd, 0L, 1)`

Il est possible d'initialiser plusieurs entrées-sorties sur un même fichier; chacune disposant de son propre index de position, elles se déroulent de façon indépendante. Voici un exemple illustrant le fonctionnement de l'appel `lseek`, dans lequel deux entrées-sorties sont ouvertes en mode lecture sur un même fichier.

```
position.c
#include <stddef.h>
#include <stdio.h>
#include <fcntl.h>

#include "es_tests.h"
#include "concatener.h"

static void afficher_les_positions(char *, int, int);

main(int argc, char *argv[])
{
    char *nom = ((argc == 2) ? argv[1]
                 : concatener(argv[0], ".test", NULL));
    int fd1 = open(nom, O_RDONLY);
    int fd2 = open(nom, O_RDONLY);
```

```
    if (fd1 == -1 || fd2 == -1)
    {
        perror(concatener(argv[0], ":", nom, NULL));
        exit(1);
    }

    afficher_les_positions("Positions initiales", fd1, fd2);

    lseek(fd1, 5L, 0);
    lseek(fd2, 1L, 0);
    afficher_les_positions("Apres les deplacements", fd1, fd2);

    es_lecture_sur(fd1, "fd1");
    es_lecture_sur(fd2, "fd2" );
    afficher_les_positions("Apres les lectures", fd1, fd2);
}

static void afficher_les_positions(char *s, int fd1, int fd2)
{
    printf("%s\n", s);
    es_afficher_position(fd1, "fd1");
    es_afficher_position(fd2, "fd2");
}
========================================== position.c ======
```

Ce programme utilise des fonctions de test contenues dans le fichier suivant :

```
======== es_tests.c ========================================
#include "es_tests.h"

void es_afficher_position(int fd, char *message)
{
    printf(" - %s: index de position en %d\n",
           message, lseek(fd, 0L, 1));
}

void es_lecture_sur(int fd, char *message)
{
    char c;
    int n = read(fd, &c, sizeof c);

    if (n == 0)
        printf(" EOF sur %s\n", message);
    else if (n < sizeof c)
        perror(message);
    else
        printf(" %s: lu \'%c\'\n", message, c);
}

void es_ecriture_sur(int fd, char c, char *message)
{
    int n = write(fd, &c, sizeof c);

    if (n < sizeof c)
        perror(message);
```

```
    else
        printf(" %s: ecrit \'%c\'\n", message, c);
}
═══════════════════════════════════════ es_tests.c ═══════════
```

On peut voir qu'il est ainsi possible de lire ce fichier en deux endroits différents, sans que les deux entrées-sorties n'interfèrent.

```
$ cat ALPHABET
abcdefghijklmnoprstuvwxyz
$
$ position ALPHABET
Positions initiales
 - fd1: index de position en 0
 - fd2: index de position en 0
Apres les deplacements
 - fd1: index de position en 5
 - fd2: index de position en 1
 fd1: lu 'f'
 fd2: lu 'b'
Apres les lectures
 - fd1: index de position en 6
 - fd2: index de position en 2
$
```

Lors d'une ouverture en écriture avec l'option O_APPEND, l'index de position est systématiquement amené à la fin du fichier avant chaque opération d'écriture. L'exemple suivant illustre l'utilisation de l'option O_APPEND en écriture.

```
═══════════ append.c ══════════════════════════════════════
#include <stddef.h>
#include <stdio.h>
#include <fcntl.h>

#include "es_tests.h"
#include "concatener.h"

main(int argc, char *argv[])
{
    char *nom = ((argc == 2) ? argv[1]
                 : concatener(argv[0], ".test", NULL));
    int fd = open(nom, O_RDWR|O_APPEND);

    if (fd == -1)
    {
        perror(concatener(argv[0], ":", nom, NULL));
        exit(1);
    }

    es_afficher_position(fd, "Position initiale");

    lseek(fd, 16L, 0);
    es_afficher_position(fd, "Positionnement");
```

```
    es_lecture_sur(fd, "Lecture");
    es_afficher_position(fd, "Apres lecture");

    es_ecriture_sur(fd, '#', "Ecriture 1");
    es_afficher_position(fd, "Apres ecriture 1");

    lseek(fd, -1L, 1);
    es_afficher_position(fd, "Repositionnement");

    es_ecriture_sur(fd, '\n', "Ecriture 2");
    es_afficher_position(fd, "Apres ecriture 2");

    close(fd);
}
```

append.c

L'index de position courante est systématiquement ramené sur le dernier caractère du fichier avant une opération d'écriture.

```
$ cat ALPHABET
abcdefghijklmnoprstuvwxyz
$
$ append ALPHABET
 - Position initiale: index de position en 0
 - Positionnement: index de position en 16
 Lecture: lu 'r'
 - Apres lecture: index de position en 17
 Ecriture 1: ecrit '#'
 - Apres ecriture 1: index de position en 27
 - Repositionnement: index de position en 26
 Ecriture 2: ecrit '
'
 - Apres ecriture 2: index de position en 28
$
$ cat ALPHABET
abcdefghijklmnoprstuvwxyz
#
$
```

10.3 Contrôle des entrées-sorties

10.3.1 Redirection des entrées-sorties standard

Nous avons vu que toute unité d'entrée-sortie, qu'elle soit associée à une unité physique comme un disque dur, ou logique comme une fenêtre, est identifiée par un fichier spécial, et que pour le programmeur, c'est le même ensemble d'appels système (`open`, `read`, `write`, `close`...) qui permet d'accéder à un fichier régulier, à un répertoire, à un terminal, aux disques, à la mémoire, etc.

Par exemple, si un processus a la permission d'écrire sur la console opéra-

teur, il lui est possible d'y faire afficher un message de la façon suivante :

```
int fd = open("/dev/console", O_RDONLY);

write(fd, message, strlen(message));
close(fd);
```

La suite d'instructions

```
int fd = open("/dev/kmem", O_RDONLY);

lseek(fd, (long) depl, 0);
read(fd, buf, sizeof buf);
```

permet de lire le contenu de la mémoire d'exécution du noyau à partir du *depl+1*^ème^ octet.

Le comportement des entrées-sorties sur un fichier spécial et sur un fichier régulier étant fondamentalement le même, il est facile de substituer une unité d'entrée-sortie par une autre. Le mécanisme le plus classique consiste à redéfinir l'entrée et/ou la sortie standard d'un processus. Ainsi, lorsqu'on tape sous un shell la commande

```
cat > texte
```

celui-ci effectue successivement :

1) la fermeture de l'entrée-sortie associée au descripteur 1;
2) la création d'un fichier de nom `texte`, par exemple au moyen de l'appel `creat`; une entrée-sortie est initialisée en mode écriture sur ce nouveau fichier et associée au premier descripteur libre, en l'occurrence le numéro 1;
3) le lancement du processus `cat` qui hérite de l'environnement d'entrée-sortie du processus qui l'a créé, c'est-à-dire :
   - 0 ouvert en lecture sur l'entrée standard,
   - 1 ouvert en écriture sur le fichier de nom `texte`,
   - 2 ouvert en écriture sur la sortie erreur standard.

Lors de son exécution, le processus `cat` effectue ses écritures sur sa sortie standard, sans savoir qu'elles sont détournées vers un fichier. On peut évidemment, par une opération de même nature sur le descripteur 0, rediriger l'entrée standard.

L'exemple suivant est un programme de trois instructions effectuant une redirection de sa sortie standard vers un fichier, puis lançant la commande DATE (1) dans ce nouvel environnement.

```
═══════ redirection.c ═══════════════════════════
#include <stdlib.h>

main()
{
    close(1);
    creat("date.resultat", 0644);
    system("date");
}
═══════════════════════════════ redirection.c ═══════
```

On vérifie tout d'abord qu'il n'existe pas de fichier **date.resultat** avant l'exécution de la commande **redirection**. Une fois celle-ci terminée, un fichier de nom **date.resultat** a été créé; il contient le résultat de la commande **date** lancée par **redirection**.

```
$ ls date.resultat
date.resultat not found
$
$ date
Thu Aug 10 14:09:00 PDT 1989
$ redirection
$
$ cat date.resultat
Thu Aug 10 14:09:03 PDT 1989
$
```

10.3.2 Détection de fin de fichier

Nous allons revenir sur le programme **recopie**, présenté page 341, recopiant son entrée standard sur sa sortie standard. Nous avions mentionné comment l'utiliser pour lister son propre source, par redirection de son entrée standard. La terminaison du programme est provoquée par la détection d'une fin de fichier, c'est-à-dire dans ce cas par la lecture du dernier caractère du fichier.

Le programme peut également être invoqué sans redirection :

```
$ recopie
abcdefg
abcdefg
123456789
123456789
il repete tout ce que je dis...
il repete tout ce que je dis...
C-d
$
```

Dans ce cas toutes les lignes entrées au clavier sont recopiées sur l'écran. Pour terminer l'exécution du programme, il suffit de taper le caractère C-d en début de ligne, qui est interprété par le pilote d'entrées-sorties du terminal (le *driver tty*) comme le caractère EOF.

La commande **recopie** peut aussi être lancée avec son entrée standard redirigée au moyen d'un tube sur la sortie standard d'un autre processus :

```
$ date | recopie
Fri Aug 11 15:24:33 PDT 1989
$
```

La détection de fin de fichier obéit ici à troisième mécanisme :

- plus de données à lire dans le tube,
- aucune entrée-sortie ouverte en écriture sur le tube.

En résumé, dans les trois cas que nous venons d'évoquer, la cause de la détection de fin de fichier est différente. Par contre, l'effet est le même : l'appel `read` retourne la valeur zéro.

Cette cohérence permet l'indépendance de comportement des programmes en lecture, vis à vis de l'unité logique sur laquelle ils effectuent les lectures. Le test de fin de fichier s'écrit toujours de la même façon :

```
if (read(fd, ...) == 0)
```

Les entrées-sorties de haut niveau gèrent un caractère `EOF` défini dans le fichier `stdio.h`. Lorsqu'on se limite à l'utilisation des entrées-sorties de haut niveau, on a l'illusion que les fichiers texte sont terminés par ce caractère spécial. En réalité, ce caractère n'a pas de signification pour les entrées-sorties de bas niveau, car il ne correspond à rien dans le noyau UNIX. Il est simulé par les fonctions de lecture de la bibliothèque standard. Par exemple, une façon élémentaire de récrire la fonction `getchar` pourrait être :

getchar.c

```
#include <stdio.h>

#ifdef getchar
# undef getchar
#endif

int getchar()
{
    char c;

    if (read(0, &c, 1) == 0)
        return EOF;
    return c;
}
```

getchar.c

Remarque 33 *Le symbole* `getchar` *est en général défini comme une pseudo-fonction dans le fichier en-tête* `<stdio.h>`*. Or ce fichier est inclus pour bénéficier de la définition du symbole* `EOF`*. Il est donc nécessaire d'annuler la définition de ce symbole avant de le définir comme un identificateur de fonction.*

10.3.3 Lectures non bloquante

`<fcntl.h>`

Il est possible de modifier le mode de fonctionnement d'une entrée-sortie déjà ouverte. Un mécanisme particulièrement intéressant est celui des lectures non bloquantes. Il concerne plus particulièrement les lectures sur un terminal ou dans un tube. Par défaut, une requête de lecture de n_1 caractères sur ces unités d'entrées-sorties sera bloquée si aucun caractère n'est disponible. Lorsqu'il y a un nombre n_2 de caractères valides, l'appel `read` est débloqué avec lecture de n caractères, n étant le minimum de n_1 et n_2. En mode non bloquant, si aucun caractère n'est valide, l'appel `read` retourne immédiatement la valeur -1 et la variable `errno` est positionnée à `EWOULDBLOCK`.

Ces opérations de contrôle sur une entrée-sortie sont réalisées au moyen de l'appel FCNTL(2). Son comportement est identique sous BSD et SYSTEM V. Il s'utilise avec trois arguments :

`int fcntl(int` ⟨*descripteur*⟩, `int` ⟨*commande*⟩, `int` ⟨*argument*⟩)

Les commandes sont des codées par des pseudo-constantes définies dans le fichier en-tête `<fcntl.h>`. La commande `F_GETFL` permet d'obtenir une copie de l'état courant de l'entrée-sortie référencée par ⟨*descripteur*⟩, sous la forme d'un entier B représentant un ensemble de booléens. Cette commande n'utilise pas d'argument. Il est possible de modifier l'état codé par B par des opérations de masquage au moyen de masques prédéfinis. Par exemple B `|= O_NDELAY` positionne dans B le mode non bloquant *no delay*. La commande `F_SETFL` permet de recopier le paramètre ⟨*argument*⟩ comme nouvel état courant de l'entrée-sortie. Ainsi, le passage en mode non bloquant peut s'écrire:

```
{
    int mode = fcntl(fd, F_GETFL, 0);
    mode |= O_NDELAY;
    fcntl(fd, F_SETFL, mode);
}
```

ou encore

```
fcntl(fd, F_SETFL, fcntl(fd, F_GETFL, 0) | O_NDELAY);
```

Il est ainsi possible, dans un programme effectuant parallèlement des lectures et des traitements, de tester s'il y a des caractères à lire, et si cela n'est pas le cas, de poursuivre les calculs au lieu de rester bloqué en lecture.

Le programme suivant illustre la mise en œuvre de ce mécanisme. Après avoir positionné l'entrée-standard en mode non bloquant, il effectue une lecture au clavier toutes les trois secondes. L'attente est gérée au moyen de l'appel SLEEP(2) qui endort le processus durant un nombre de secondes qu'il reçoit en paramètre. Lorsqu'il y a des caractères présents dans le tampon d'entrée, ils sont lus par `read`; si ce n'est pas le cas, `read` retourne -1. La fonction `hms`, définie page 238, qui permet d'obtenir le temps en heures, minutes et secondes, est utilisée pour visualiser la fréquence des lectures.

```
======== es_controle.c ========
#include <setjmp.h>
static jmp_buf contexte;

#include <signal.h>
#include <fcntl.h>
#include <stdio.h>

#define DELAI 3
#define TBUF  8

static void retour();

main()
{
    int h, m, s;

    signal(SIGINT, retour);
```

```
    signal(SIGQUIT, retour);
    signal(SIGTERM, retour);
    signal(SIGTSTP, retour);

    if (setjmp(contexte) == 0)
    {
        fcntl(0, F_SETFL, fcntl(0, F_GETFL) | O_NDELAY);
        for (;;)
        {
            char buffer[TBUF-1];
            int n;

            sleep(DELAI);
            hms(&h, &m, &s);
            n = read(0, buffer, sizeof buffer);
            if (n == 1 && buffer[0] == '\n')
                break;

            printf("  [%d:%d:%d] read retourne %d\n",
                   h, m, s, n);
            if (n > 0)
            {
                buffer[n] = '\0';
                printf("  lu: \"%s\"\n", buffer);
            }
        }
    }
    fcntl(0, F_SETFL, fcntl(0, F_GETFL) & ~O_NDELAY);
}

static void retour()
{
    longjmp(contexte, 1);
}
```
es_controle.c

Il est très important de restaurer l'état initial de l'entrée-sortie associée au terminal avant de terminer le programme. En effet, le mode de fonctionnement du pilote de terminal ne fait pas partie de l'environnement du processus mais de celui du pilote lui-même. Il est par conséquent commun à tous les processus utilisant cette entrée-sortie, y compris le shell attaché à la console. On remarquera l'utilisation du mécanisme de sauvegarde de contexte (fonctions `setjmp` et `longjmp` afin de gérer une sortie correcte sur réception de signaux. La sauvegarde de contexte est traitée dans la section 7.9, et les signaux dans la section 12.2. Un signal est un message envoyé à un processus, et qui par défaut, provoque sa terminaison. Nous verrons plus loin que la fonction **SIGNAL(3)** permet de modifier ce comportement par défaut. Dans une première lecture, on pourra ignorer les lignes de récupération du signal :

```
signal(SIGINT, retour);
signal(SIGQUIT, retour);
signal(SIGTERM, retour);
signal(SIGTSTP, retour);
```

et celle de son traitement :

```
if (setjmp(contexte) == 0)
{

}
```

et

```
static void retour()
{
    longjmp(contexte, 1);
}
```

Lors de la session suivante, on peut vérifier que la lecture est effectuée de manière non blocante toutes les trois secondes. Les lectures se font par paquets de sept caractères, incluant le caractère `\n` en fin de ligne, et le nombre de caractères réellement lus est affiché. La sortie de boucle se fait sur la saisie d'une chaîne vide.

```
$ es_controle
  [15:8:17] read retourne -1
  [15:8:20] read retourne -1
abcdefghijkl
  [15:8:23] read retourne 7
  lu: "abcdefg"
  [15:8:26] read retourne 6
  lu: "hijkl
"
  [15:8:29] read retourne -1
C-m
$
```

10.3.4 Interactions entre haut niveau et bas niveau

Il est tentant de combiner les entrées-sorties de haut niveau et de bas niveau. On bénéficie ainsi de la sophistication des premières (par exemple le formatage) et de l'efficacité des secondes. Cela pose cependant quelques problèmes de synchronisation.

Les entrées-sorties de bas niveau n'étant pas mémorisées, elles sont physiquement effectuées à chaque appel, ce qui n'est pas le cas des entrées-sorties de haut niveau. L'erreur la plus classique consiste à combiner des écritures mémorisées (`printf` par exemple) avec des lectures immédiates au moyen de l'appel système `read`. Considérons le programme `trivial_es.c` suivant :

```
trivial_es.c
main()
{
    char c;

    printf("Taper un caractere: ");
    read(0, &c, 1);
```

```
    printf("Caractere lu '%c'\n", c);
}
```
trivial_es.c

On peut s'attendre à ce qu'une exécution de ce programme se déroule de la façon suivante (pour plus de lisibilité, les information affichées par le programme sont visualisées caractères penchés) :

```
$ EsHautbas
Taper un caractere: X
Caractere lu 'X'
$
```

En réalité, l'écriture produite par le premier `printf` ne s'effectue physiquement qu'après l'exécution de l'appel `read`; la chaîne `"Taper un caractère: "` est mémorisée et le tampon de mémorisation vidé sur le caractère `'\n'` de la seconde écriture. L'exécution du programme s'effectue de la façon suivante :

```
$ EsHautbas
X
Taper un caractere: Caractere lu 'X'
$
```

On peut résoudre se problème au moyen de la fonction `fflush` permettant de forcer le vidage du tampon (voir 7.1.6).

Lorsque cela n'est pas nécessaire, on évitera de combiner les entrées-sorties de haut et de bas niveaux.

10.4 Configuration des pilotes de terminaux

10.4.1 Fonctionnement d'un pilote tty

Par défaut, lorsqu'un programme effectue une lecture sur son entrée standard, les caractères tapés au clavier sont dans un premier temps mémorisés dans un tampon local au pilote afin de permettre l'édition de la ligne en cours de saisie. En effet, certains caractères sont considérés par le pilote comme des caractères spéciaux et sont interprétés au lieu d'être transmis à l'appel `read`. Par exemple les caractères *erase* et *kill*, initialisés respectivement à *backspace* ou *delete* et à `C-u` provoquent l'annulation, l'un du dernier caractère tapé et l'autre de toute la ligne. Le nombre et la signification de ces caractères peuvent varier suivant les systèmes, mais les mécanismes de base sont toujours les mêmes. On peut connaître la liste des caractères interprétés par le pilote *tty* au moyen de la commande STTY(1).

```
$ stty all
speed 9600 baud, 24 rows, 80 columns; evenp
line = 2
-inpck imaxbel
iexten crt
erase kill werase rprnt flush lnext susp  intr quit stop  eof
```

```
^H   ^U  ^W   ^R   ^O   ^V   ^Z/^Y ^C   ^\   ^S/^Q ^D
$
```

L'option `all` est propre au système UNIX BSD; l'option correspondante sous SYSTEM V est `-a`. Certaines des fonctionnalités BSD ne sont pas définies sous SYSTEM V et réciproquement. Parmi les caractères spéciaux listés par l'exécution de la commande `stty`, on trouve :

`erase`	: effacement du dernier caractère
`kill`	: annulation de la ligne
`eof`	: fin de fichier
`intr`	: interruption de la commande en cours
`quit`	: idem, avec production d'un fichier `core`
`stop`	: contrôle de flux
`werase`	: effacement du dernier mot (BSD)

La figure 10.3 illustre le fonctionnement d'une lecture de caractères sur l'entrée standard dans sa configuration par défaut :

- les caractères tapés au clavier (1) sont récupérés par le pilote *tty* et stockés dans une mémoire tampon;
- chaque caractère est en même temps affiché par le pilote sur l'écran (2) ;
- la fonction `read` du processus est reliée, à travers le descripteur 0, à la fonction de lecture spécifique au terminal (3); chaque appel à `read` lit les caractères valides stockés dans le tampon du pilote *tty*.

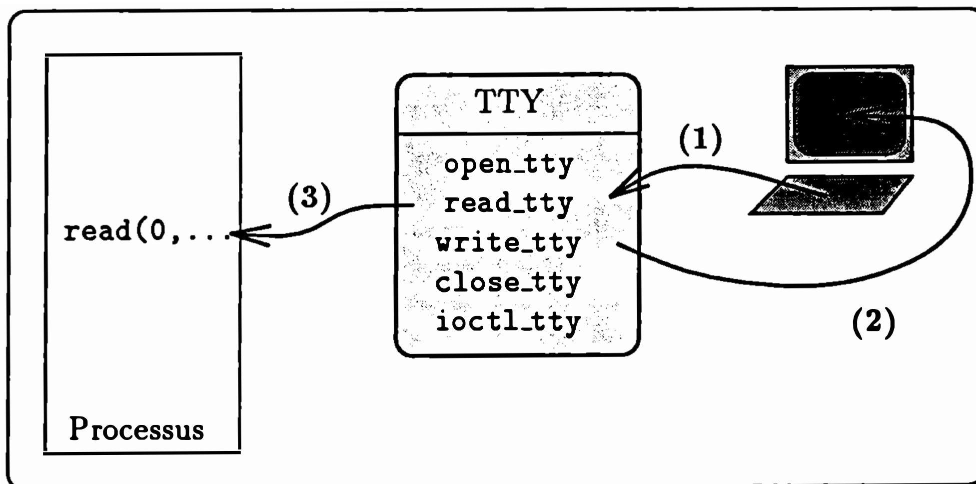


Figure 10.3: Fonctionnement d'une lecture au clavier

Par défaut, les caractères entrés au clavier sont validés au moyen des caractères spéciaux `C-m` (ou touche RETURN) et `C-j` (ou touche LINEFEED) qui sont alors convertis par le pilote *tty* en un seul et même caractère, le caractère `'\n'`. Si `read` demande moins de caractères qu'il n'y en a de visibles dans le buffer, les caractères restants sont conservés pour la prochaine lecture. Si le nombre de caractères demandé est supérieur au nombre de caractères disponibles, la lecture est effectuée tout de même sur les caractères présents. Une requête de lecture n'est bloquante que lorsque le buffer ne contient aucun caractère actif. La fonction `read` renvoie le nombre de caractères réellement lus.

On a parfois besoin de saisir un caractère au moment précis où il a été tapé au clavier (on parle parfois de lecture *caractère par caractère*). C'est par exemple le mode de fonctionnement d'un éditeur de texte vidéo comme `emacs` ou `vi` : il n'est pas nécessaire de taper RETURN après chaque caractère... Le mode lecture caractère par caractère est un mode de fonctionnement particulier du pilote d'entrées-sorties *tty*. Il existe d'autres possibilités de paramétrage, comme la suppression de l'écho des caractères tapés (l'écriture étiquetée **(2)** sur la figure 10.3), la désactivation du traitement des caractères spéciaux, etc.

Le passage en mode caractère par caractère s'effectue différemment sous BSD et SYSTEM V. Sous UNIX BSD, il existe un mode spécifique appelé le mode **cbreak**. Sous UNIX SYSTEM V, il existe un mode plus général s'obtenant en désactivant le mode **canonique** d'interprétation des caractères spéciaux; il est alors possible de préciser un nombre NC de caractères, et un délai DL exprimé en dixièmes de seconde, à partir desquels l'activation s'effectue automatiquement (si le nombre de caractères et le délai sont égaux à 1, on retrouve le mode *cbreak* de BSD).

La façon la plus simple de contrôler le mode de fonctionnement du pilote est d'utiliser au niveau de l'interprète de commandes, la commande **STTY(1)**. Par exemple, la suppression de l'écho s'obtient par

```
stty -echo
```

et sa réactivation par

```
stty echo
```

De cette façon, il est possible de lancer une commande ⟨*com*⟩ en mode lecture caractère par caractère en tapant sous BSD

```
$ stty cbreak
$ ⟨com⟩
$ stty -cbreak
```

et sous SYSTEM V

```
$ stty eof ^A eol ^A -icanon
$ ⟨com⟩
$ stty eof ^D icanon
```

Sous SYSTEM V, lorsque le mode canonique est désactivé, les paramètres NC et DL sont rangés dans les champs `eof` et `eol` du pilote *tty*. Sur l'exemple, ils reçoivent la valeur du caractère `C-a`, c'est-à-dire 1.

Le principal inconvénient de cette méthode est de ne pouvoir repasser en mode normal pendant l'exécution de la commande `com`. On peut alors utiliser la fonction **SYSTEM(3)** pour lancer la commande `stty` directement depuis le programme, par exemple sous BSD :

```
system("stty cbreak");
lecture();
system("stty -cbreak");
```

Il est enfin possible de paramétrer un pilote *tty* directement au moyen de l'appel système **IOCTL(2)**. Cet appel s'emploie avec trois paramètres :

1) un descripteur de fichier;
2) une commande, définie sous la forme d'une pseudo-constante;

3) un paramètre optionnel, dépendant de la commande.

La syntaxe est :

`int ioctl(int ⟨descripteur⟩, int ⟨commande⟩, int ⟨argument⟩)`

Les commandes spécifiques au pilote de terminal sont décrites dans le chapitre **TTY(4)** du manuel UNIX BSD et **TERMIO(7)** du manuel UNIX SYSTEM V.

Chaque communication vers un terminal est décrite par un ensemble de paramètres internes au pilote, qu'il est possible d'obtenir au moyen de l'appel **IOCTL(2)**, puis de modifier avec des opérations booléennes de masquage, et enfin de retourner au pilote, toujours au moyen de l'appel `ioctl`. On reconnaîtra le principe d'utilisation de l'appel `fcntl` présenté en 10.3.3.

Comme nous l'avons déjà dit, les pilotes *tty* sont différents sur BSD et sur SYSTEM V. Nous présentons rapidement dans les sections suivantes deux fichiers en-tête permettant de mettre en œuvre simplement et de manière uniforme les principales fonctionnalités du paramétrage de pilote *tty* sous chacun des deux systèmes. Le lecteur pourra ne consulter que la section se rapportant à la version du système qu'il utilise.

Les deux versions définissent le même ensemble de pseudo-fonctions:

- `savetty()` : sauvegarde de l'état courant;
- `_savetty(⟨s⟩)` : idem, mais dans une variable ⟨*s*⟩ de type `tty_svg` – cette fonction ou la précédente doivent être appelée avant toute autre;
- `resetty()` : restauration de l'état sauvegardé par `savetty`;
- `_resetty(⟨s⟩)` : restauration de l'état sauvegardé dans ⟨*s*⟩;
- `cbreak()` : activation du mode caractère par caractère;
- `nocbreak()` : désactivation du mode caractère par caractère;
- `raw()` : activation du mode *cru* (aucun prétraitement des caractères);
- `noraw()` : désactivation du mode *cru*;
- `echo()` : activation de l'écho;
- `noecho()` : désactivation de l'écho;
- `nl()` : activation du traitement CR/LF;
- `nonl()` : désactivation du traitement CR/LF;
- `ff0()`, `ff1()` : paramétrage de la temporisation sur saut de page.

De plus les pseudo-constantes `ERASE` et `KILL` sont associées aux caractères d'annulation du dernier caractère entré et de la ligne courante.

10.4.2 Mise en œuvre sous Bsd

`<sgttyb.h>`

Nous allons considérer ici une forme simple d'interaction avec le pilote *tty*. Elle est supportée par les implémentations récentes de BSD, bien que le mécanisme ait été enrichi. Dans cette version, les paramètres du pilote *tty* BSD sont décrits par une structure appelée

`struct sgttyb`

et définie dans le fichier en-tête `<sgttyb.h>`. Un des champs de cette structure, le champ `sg_flags`, représente une liste de booléens également définis dans le même fichier en-tête, dont les principaux sont :

– CBREAK : mode caractère par caractère.
– ECHO : écho des caractères tapés.
– RAW : tous les caractères spéciaux sont ignorés.
– CRMODE : chacun des caractères `\r` et `\n` est converti en la suite CR/LF lors de son affichage sur l'écran.

Pour modifier les paramètres de fonctionnement du pilote *tty*, il convient tout d'abord de déclarer une structure de type `struct sgttyb`, par exemple

```
struct sgttyb terminal;
```

La récupération des paramètres courants du pilote *tty* correspond à la commande TIOCGETP qu'on met en œuvre de la façon suivante :

```
ioctl(0, TIOCGETP, &terminal)
```

On suppose ici que le descripteur 0 est associé à l'entrée standard par défaut.

Par des opérations classiques de masquage, il est possible de modifier les booléens contenus dans le champ `sg_flags`; par exemple, la suppression de l'écho et le passage en mode *cbreak* s'écrivent :

```
terminal.sg_flags |= CBREAK;
terminal.sg_flags &= ~ECHO;
```

Enfin, la nouvelle description est retournée au pilote *tty* au moyen de la commande TIOCSETP :

```
ioctl(0, TIOCSETP, &terminal)
```

On se reportera au manuel UNIX pour une description complète des commandes et paramètres

Le fichier en-tête suivant permet de contrôler les principales fonctionnalités d'un pilote *tty* sous UNIX BSD.

```
========= tty_bsd.h ==========================================
#ifndef TTY_BSD_H
#define TTY_BSD_H

/*************************************************************\
*     Pseudo-fonctions de configuration tty pour BSD     *
\*************************************************************/

#include <sgtty.h>
struct sgttyb _ttyb_;

/* Pour la sauvegarde par defaut de l'etat du tty */
int    _ttyb_svg_;
typedef int tty_svg;

/*
 * Avant toute chose, appeler savetty() ou _savetty()
 */
#define savetty()    (_savetty (_ttyb_svg_))
#define resetty()    (_resetty (_ttyb_svg_))

#define _savetty(fl) (ioctl (0, TIOCGETP, &_ttyb_),\
                      fl = _ttyb_.sg_flags)
#define _resetty(fl) (_ttyb_.sg_flags = fl, \
                      ioctl(0, TIOCSETP, &_ttyb_))
```

```
/* Activation/desactivation d'un mode */
#define termset(mode)     (_ttyb_.sg_flags |= (mode), \
                           ioctl(0, TIOCSETP, &_ttyb_))
#define termuns(mode)     (_ttyb_.sg_flags &= ~(mode), \
                           ioctl(0, TIOCSETP, &_ttyb_))

/* Activation/desactivation du mode cbreak */
#define cbreak()     termset(CBREAK)
#define nocbreak()   termuns(CBREAK)

/* Activation/desactivation du mode raw */
#define raw()        termset(RAW)
#define noraw()      termuns(RAW)

/* Activation/desactivation de l'\'echo */
#define echo()       termset(ECHO)
#define noecho()     termuns(ECHO)

/* Conversion de CR et LF en CR/LF */
#define nl()         termset(CRMOD)
#define nonl()       termuns(CRMOD)

/* Delai apres un saut de page */
#define ff0()        termuns(FF1)
#define ff1()        termset(FF1)

/* Caracteres speciaux */
#define ERASE    _ttyb_.sg_erase
#define KILL     _ttyb_.sg_kill

#endif  /* TTY_BSD_H */
```

tty_bsd.h

La pseudo-fonction `savetty` initialise la structure de description des paramètres et sauvegarde l'état du terminal. Elle doit être utilisée en premier. La fonction `resetty` restaure l'état du terminal sauvegardé par `savetty`.

Remarque 34 *Sous* UNIX V7, *existaient les appels* STTY(2) *et* GTTY(2) *qui sont des versions simplifiées de* `ioctl`, *spécialisées dans la configuration des pilotes* tty. *Pour des raisons de compatibilité, ces appels ont été conservés dans les différentes versions de* UNIX BSD. *Il est cependant préférable d'utiliser l'appel* `ioctl`.

10.4.3 Mise en œuvre sous System V

`<termio.h>`

Les paramètres du pilote *tty* SYSTEM V sont décrits par une structure appelée

`struct termio`

et définie dans le fichier en-tête `<termio.h>` Les champs de cette structure sont des listes de booléens décrivant le comportement du pilote en lecture et en écriture, et un vecteur de caractères définissant les différents caractères spéciaux qu'il interprète en mode canonique. Pour modifier les paramètres de

fonctionnement du pilote *tty*, il convient tout d'abord de déclarer une structure de type `struct termio`, par exemple

```
struct termio terminal;
```

La récupération des paramètres courants du pilote *tty* correspond à la commande TCGETA qu'on met en œuvre de la façon suivante :

```
ioctl(0, TCGETA, &terminal)
```

On suppose ici que le descripteur 0 est associé à l'entrée standard par défaut. Par des opérations classiques de masquage, il est possible de modifier les booléens contenus dans les différents champs; par exemple la suppression de l'écho s'écrit :

```
terminal.c_lflag |=  (ECHO | ECHOE)
```

Enfin, la nouvelle description est retournée au pilote *tty* au moyen de la commande TCSETA :

```
ioctl(0, TCSETA, &terminal)
```

On se reportera au chapitre **TERMIO (7)** du manuel UNIX pour une description complète des commandes et paramètres.

Le fichier en-tête suivant permet de contrôler les principales fonctionnalités d'un pilote *tty* sous UNIX SYSTEM V.

```
============ tty_sv.h ==========================================
#ifndef TTY_SV_H
#define TTY_SV_H

/**************************************************************\
*    Pseudo-fonctions de configuration tty pour System V     *
\**************************************************************/

#include <termio.h>
struct termio _termiob;

/* Pour la sauvegarde par defaut de l'etat du tty */
struct termio    __termiob;
unsigned short   __rawi, __rawcs;
char             __eof, __eol, __raweof, __raweol;
typedef struct termio tty_svg;

/*
 * Avant toute chose, appeler savetty() ou _savetty
 */
#define savetty()     (_savetty(__termiob))
#define resetty()     (_resetty(__termiob))

#define _savetty(b)  (ioctl(0, TCGETA, &_termiob), b=_termiob)
#define _resetty(b)  (_termiob=b, ioctl(0, TCSETA, &_termiob))

/* Gestion des lectures caractere par caractere */
#define cbreak()    (__eof= _termiob.c_cc[VEOF],\
                     __eol= _termiob.c_cc[VEOL],\
                     _termiob.c_cc[VEOF] = \
                     _termiob.c_cc[VEOL] = 1,\
                     _termiob.c_lflag &= ~ICANON,\
```

```
                    ioctl(0, TCSETA, &_termiob))

#define nocbreak()  (_termiob.c_cc[VEOF]= __eof,\
                    _termiob.c_cc[VEOL]= __eol,\
                    _termiob.c_lflag |= ICANON,\
                    ioctl(0, TCSETA, &_termiob))

/*
 * Mode raw : - lecture caractere par caractere
 *            - caracteres lus en mode cru:
 *                - 8 bit,
 *                - ^C, ^D, ^U,... non interpretes,
 *                - ...
 */

#define raw()  (__rawi = _termiob.c_iflag,\
               _termiob.c_iflag = 0,\
               _termiob.c_oflag &= ~OPOST,\
               __rawcs = _termiob.c_cflag & CSIZE,\
               _termiob.c_cflag |= CSIZE,\
               _termiob.c_cflag &= ~PARENB,\
               _termiob.c_lflag &= ~(ICANON | ISIG),\
               __raweof = _termiob.c_cc[VEOF],\
               __raweol = _termiob.c_cc[VEOL],\
               _termiob.c_cc[VEOF]= _termiob.c_cc[VEOL]= 1,\
               ioctl(0, TCSETA, &_termiob))

#define noraw() (_termiob.c_iflag = __rawi,\
                _termiob.c_oflag |= OPOST,\
                _termiob.c_cflag &= ~CSIZE,\
                _termiob.c_cflag |= __rawcs,\
                _termiob.c_cflag |= PARENB,\
                _termiob.c_lflag |= (ICANON | ISIG),\
                _termiob.c_cc[VEOF] = __raweof,\
                _termiob.c_cc[VEOL] = __raweol,\
                ioctl(0, TCSETA, &_termiob))

/* Activation/desactivation de l'echo */
#define echo()      (_termiob.c_lflag |= (ECHO | ECHOE),\
                    ioctl(0, TCSETA, &_termiob))
#define noecho()    (_termiob.c_lflag &= ~(ECHO | ECHOE),\
                    ioctl(0, TCSETA, &_termiob))

/* Conversion de CR et LF en CR/LF */
#define nl()        (_termiob.c_oflag &= ~ONLCR,\
                     ioctl(0, TCSETA, &_termiob))
#define nonl()      (_termiob.c_oflag |= ONLCR,\
                     ioctl(0, TCSETA, &_termiob))

/* Delai apres un saut de page */
#define ff0()       (_termiob.c_oflag &= ~FF1,\
                     ioctl(0, TCSETA, &_termiob))
#define ff1()       (_termiob.c_oflag |= FF1,\
                     ioctl(0, TCSETA, &_termiob))

/* Caracteres speciaux */
#define ERASE   (_termiob.c_cc[VERASE])
#define KILL    (_termiob.c_cc[VKILL])
```

```
#endif  /* TTY_SV_H */
```
tty_sv.h

La version SYSTEM V de ce fichier est plus complexe que la version BSD car le fonctionnement du pilote *tty* est plus complet. Le mode *cbreak* est un cas particulier d'un ensemble de modes de fonctionnement possibles : lorsque le mode canonique est désactivé, la validation des caractères tapés se fait après la lecture d'un certain nombre de caractères et au bout d'un délai exprimé en dixièmes de seconde.

Le mode canonique correspond au booléen `ICANON` du champ `c_lflag`; les caractères `EOF` et `EOL` du vecteur de caractères `c_cc` contiennent respectivement le nombre minimum de caractères à lire, et le délai de temps. Une lecture non bloquante peut être obtenue avec les valeurs :

```
termiob.c_cc[EOF] = 0
termiob.c_cc[EOL] = 1
```

10.4.4 Un exemple d'utilisation

Voici maintenant un exemple de programme utilisant ces fonctions; il s'agit d'une fonction lisant un entier dans une grille d'écran (simplifiée à l'extrême). Le curseur est bloqué dans une zone correspondant à la taille maximale de l'entier à lire, et on ne peut en sortir ni en tapant un nombre trop grand, ni en revenant trop en arrière (par effacements de caractères). C'est un exemple typique d'application devant réaliser des lectures de caractères *au vol* afin de tester chaque caractère tapé (ce doit être un chiffre) et de refuser les caractères illicites.

Nous allons commencer par définir deux fonctions permettant de gérer les déplacements horizontaux du curseur. La fonction `arriere` fait reculer le curseur de *n* colonnes vers la gauche en itérant l'écriture du caractère `\b`

```
static void arriere(int n)
{
    while (n--)
        write(1, "\b", 1);
}
```

Les caractères sont lus dans un vecteur adressé par le pointeur `buffer`; la position courante dans le vecteur est pointée par le pointeur `pbuf`. La fonction `erase` gère la suppression d'un caractère simultanément dans le vecteur et sur l'écran.

```
static int erase()
{
    if (pbuf == buffer)
        return 0;
    pbuf-- ;
    arriere(1);
    write(1, " ", 1);
    arriere(1);
    return 1;
}
```

La fonction `erase` retourne 0 si le vecteur ne contient pas de caractère lorsqu'elle est appelée, et 1 sinon. De cette façon, l'effacement de tous les caractères présents dans le vecteur peut s'écrire

```
while (erase())
    ;
```

La fonction `lire_entier` lit un entier sur *nbcarmax* caractères maximum; le curseur ne peut sortir de la zone de lecture, de largeur *nbcarmax* caractères. La fonction gère les effacements de caractères, en utilisant les caractères d'effacement propres à l'utilisateur. Elle retourne zéro si l'utilisateur a tapé une chaîne vide. La validation se fait par RETURN. L'inclusion du bon fichier de configuration du pilote *tty*, c'est-à-dire de la version BSD ou de la version SYSTEM V, peut être réalisé au moyen du fichier en-tête `tty.h`

```
tty.h
#ifndef TTY_H
#define TTY_H

#if defined SYSV

# include "tty_sv.h"

#elif defined BSD

# include "tty_bsd.h"

#endif

#endif  /* TTY_H */
tty.h
```

La compilation se fera en positionnant la pseudo-constant BSD ou SYSTEM V selon le cas. Par exemple, la compilation du fichier `lire_entier` sous système BSD peut être lancée par la commande

```
gcc -c -g -DBSD lire_entier.c
```

Le source complet du module de lecture est le suivant :

```
lire_entier.c
#include <stddef.h>
#include <stdlib.h>

#include "lire_entier.h"
#include "tty.h"

static char *buffer=NULL;
static char *pbuf;

static void arriere(int);
static int erase(void);

int lire_entier(int nbcarmax)
{
    /* sauvegarde parametres tty */
```

```
    tty_svg flg;
    char   *p;
    int    v;

    /* pas d'echo, mode cbreak */
    _savetty(flg);
    noecho();
    cbreak();

    /* Lecture jusqu'a \n ou erreur de lecture */
    buffer = malloc(nbcarmax+1);
    pbuf = buffer;
    for (;;)
    {
        int n = read(0, pbuf, 1);

        if (n != 1 || *pbuf == '\n')
            break;

        if (*pbuf == ERASE)
        {
            /* Effacement du dernier caractere lu */
            erase();
            continue;
        }
        if (*pbuf == KILL)
        {
            /* Effacement de tous les caracteres lus */
            while (erase())
                ;
            continue;
        }
        if (!isdigit(*pbuf))
        {
            /* le caractere lu n'est pas un chiffre */
            write(1, "\a", 1); /* Caractere Bell */
            continue;
        }

        if (pbuf-buffer >= nbcarmax)
        {
            /* on a lu 'nbcarmax' caracteres */
            continue;
        }
        /* Le caractere est correct, on l'affiche */
        pbuf++;
        write(1, pbuf-1, 1);
    }

    /* Conversion de la chaine lue en entier */
    for (v=0, p=buffer; p<pbuf; v = 10*v + *p++ -'0')
        ;
    /* Restauration de l'etat du terminal */
    _resetty(flg);

    free(buffer);
    return v;
}
```

```
static void arriere(int n)
{
    while (n--)
        write(1, "\b", 1);
}

static int erase()
{
    if (pbuf == buffer)
        return 0;
    pbuf-- ;
    arriere(1);
    write(1, " ", 1);
    arriere(1);
    return 1;
}
========================================== lire_entier.c ========
```

L'utilisation de cette fonction est illustrée par le programme suivant; il affiche deux étoiles séparées par cinq blancs, ramène le curseur sur le premier blanc, et demande une lecture bloquée sur cinq caractères.

```
======== lecture_controlee.c ==================================
#include <stdio.h>

#define PROMPT  "Entier sur 5 caracteres *     *\b\b\b\b\b\b"

#include "lire_entier.h"

main()
{
    printf("%s", PROMPT);
    fflush(stdout);
    printf("\nNombre lu : %d\n", lire_entier(5));
}
=================================== lecture_controlee.c ========
```

La figure 10.4 montre le résultat de l'affichage après le lancement du programme. Pendant la saisie, le curseur reste bloqué entre les deux étoiles. Les caractères illégaux sont écartés.

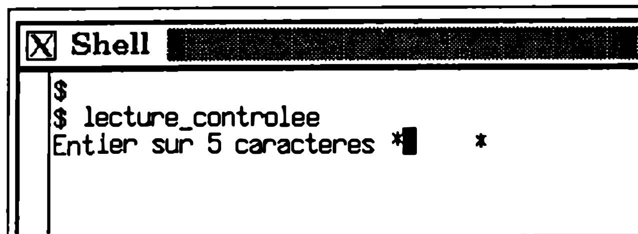


Figure 10.4: Exemple d'exécution du programme `lecture_controlee` (a)

La figure 10.5 visualise le résultat de la frappe des caractères **122←3ab45c67↵**. Le symbole ← représente le caractère *erase* (par exemple `C-h`). La seconde

occurrence du caractère 2 est effacée par le caractère *erase*, et les lettres sont ignorées. Les deux derniers chiffres sont également ignorés, le nombre maximum de chiffres demandé, ici 5, étant atteint.

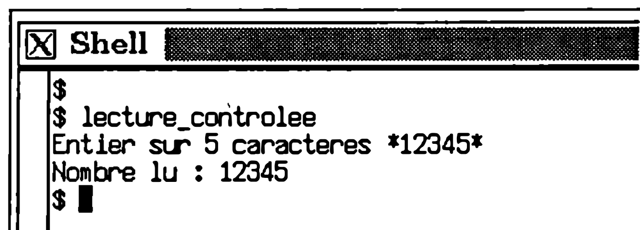


Figure 10.5: Exemple d'exécution du programme `lecture_controlee` (b)

Remarque 35 *Le programme précédent peut être interrompu par un signal, provoqué par exemple par la frappe d'un caractère d'interruption au clavier. Si c'est le cas, le terminal se retrouve dans un* trou noir *(plus d'écho et mode caractère par caractère). Pour éviter ce genre d'accident, il est important que le programme récupère ces signaux afin de gérer une sortie propre. Nous verrons cela dans le chapitre concernant les processus.*

Chapitre 11

Manipulation des fichiers

Ce chapitre est consacré à la manipulation des fichiers du système UNIX. Le terme de **fichier** recouvre différents concepts :

- les fichiers *réguliers* pouvant contenir indifféremment du texte, des données, des programmes exécutables...
- les *répertoires* contenant des références à d'autres fichiers;
- les fichiers *spéciaux* servant d'interface avec les pilotes d'entrées-sorties;
- les *liens symboliques*, *tubes nommés*, etc.

La plupart des fonctionnalités exposées sont communes à UNIX BSD et à UNIX SYSTEM V.

11.1 Organisation du système de fichiers Unix

11.1.1 Bloc de description de fichiers : inode

Chaque fichier présent dans le système de fichiers est décrit par un bloc d'informations appelé **inode**, de l'anglais *index node*. Ces informations sont de deux types :

- attributs du fichier :
  * nature du fichier (régulier, répertoire ou spécial);
  * identification du propriétaire et du groupe du fichier;
  * accès en lecture, écriture et exécution;
  * taille en octets;
  * dates de dernier accès et de dernière modification;
  * nombre de liens sur ce fichier (voir section suivante).
- contenu du fichier :
  * dans le cas d'un fichier régulier ou d'un répertoire, le chaînage des blocs de données;
  * dans le cas d'un fichier spécial, la référence au pilote d'entrées-sorties correspondant (voir chapitre 10);

* dans le cas d'un lien symbolique, le chemin du fichier dont il est le synonyme.

Les données des fichiers sont organisées en blocs, structurés au moyen de chaînages que nous ne détaillerons pas ici. Nous considèrerons simplement un tel fichier comme un vecteur d'octets, qu'il est possible de défiler séquentiellement ou en accès direct, et de taille virtuellement infinie (la taille théorique d'un fichier UNIX est actuellement de l'ordre de 16 à 64 Go.)

Chaque inode est lui-même identifié par un numéro que, pour simplifier, nous considèrerons unique dans tout le système de fichiers. Le numéro d'inode d'un fichier et ses principaux attributs peuvent être listés au moyen de la commande **LS (1)**. Sur l'exemple suivant,

```
$ ls -li preamb.sty
46545 -rw-r--r--  1 achille      1442 Aug 15 18:31 preamb.sty
$
```

le fichier `preamb.sty` est identifié par l'inode numéro 46545; il contient 1442 octets, a été modifié pour la dernière fois le 15/08 de l'année courante à 18 heures 31 minutes... Cette structure est visualisée sur la figure 11.1.

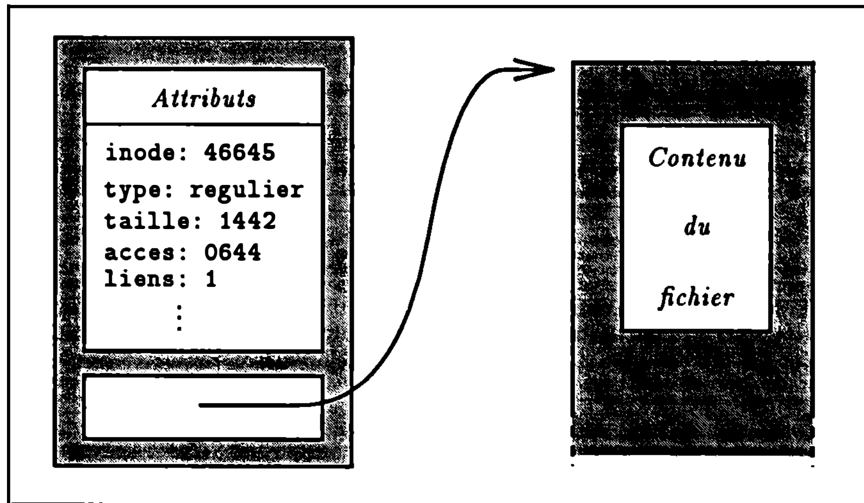


Figure 11.1: Structure d'un fichier Unix

11.1.2 Attributs d'un fichier

<sys/stat.h> <sys/types.h>

L'appel stat

L'ensemble des informations contenues dans un inode peut être obtenu au moyen de l'appel système **STAT (2)**. Cette liste d'informations est définie sous la forme d'une structure, `struct stat`, dans le fichier en-tête **`<sys/stat.h>`**. Les principaux champs de cette structure sont :

- `st_ino` : numéro de l'inode;
- `st_mode` : nature du fichier (régulier, répertoire...) et modes d'accès;

- `st_nlink` : nombre de liens sur ce fichier;
- `st_uid` : identification de l'utilisateur propriétaire du fichier;
- `st_gid` : identification du groupe propriétaire du fichier;
- `st_size` : taille du fichier en octets;
- `st_atime` : date du dernier accès au fichier : création, lecture, écriture, lecture des attributs...
- `st_mtime` : date de la dernière modification des données : création, écriture, modification d'un attribut...
- `st_ctime` : date de la dernière modification de l'état (création, écriture, changement de mode d'accès ou de propriétaire...);
- `st_dev` : identificateur du système de fichiers[1] contenant ce fichier.

Le type de chacun de ces champs est généralement prédéfini dans le fichier `<sys/types.h>`, qu'il convient par conséquent d'inclure avec le fichier `<sys/stat.h>`. L'appel `stat` s'emploie avec deux arguments, le chemin du fichier à tester et l'adresse d'une structure de type `struct stat`.

Nous en donnons un premier exemple d'utilisation à travers la commande `lsdates` listant les différentes dates d'accès et de modification d'une liste de fichiers. La date est exprimée en temps universel UNIX et convertie au moyen de la fonction de bibliothèque `localtime` (voir 7.6).

```
lsdates.c
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <time.h>

#include "concatener.h"

static void printdate(char*, time_t);

main(int argc, char **argv)
{
    char *argv0 = argv[0];

    while (++argv, --argc)
    {
        struct stat statbuf;
        char *pmessage = concatener(argv0, ":", *argv, NULL);

        if (stat(*argv, &statbuf) == -1)
        {
            perror(pmessage);
            continue;
        }
        printdate("derniere lecture/ecriture",
                  statbuf.st_atime);
        printdate("derniere ecriture",
                  statbuf.st_mtime);
        printdate("modification de l'etat",
```

[1]Sous UNIX, un système de fichiers physique est identifié par un couple de nombres appelés `major number` et `minor number`. Les systèmes de fichiers sont reliés les uns aux autres pour former une arborescence unique (il s'agit du mécanisme de montage).

```
                statbuf.st_ctime);
    }
}

static void printdate(char *s, time_t d)
{
    struct tm *tm = localtime(&d);

    printf("  %s: %02d/%02d/%02d (%02d:%02d:%02d)\n", s,
           tm->tm_mday, tm->tm_mon+1, tm->tm_year,
           tm->tm_hour, tm->tm_min, tm->tm_sec);
}
```

lsdates.c

La session suivante, effectuée sur un système UNIX BSD, montre différentes modifications des attributs de dates : création du fichier, modification de la date de dernier accès (lecture ou écriture) par `cat`, modification de la date de changement d'un attribut par `chmod`. Chaque commande entraînant une modification de date est précédée par une exécution de la commande **DATE(1)** dont le rôle est d'afficher l'heure de la modification.

```
$ date ; echo "" > TestDates
Sat May 29 00:17:26 MET DST 1993
$ lsdates TestDates
  derniere lecture/ecriture: 29/05/93 (00:17:26)
  derniere ecriture: 29/05/93 (00:17:26)
  modification de l'etat: 29/05/93 (00:17:26)
$
$ date ; cat TestDates
Sat May 29 00:17:50 MET DST 1993
$ lsdates TestDates
  derniere lecture/ecriture: 29/05/93 (00:17:50)
  derniere ecriture: 29/05/93 (00:17:26)
  modification de l'etat: 29/05/93 (00:17:26)
$
$ date ; chmod 666 TestDates
Sat May 29 00:18:15 MET DST 1993
$ lsdates TestDates
  derniere lecture/ecriture: 29/05/93 (00:17:50)
  derniere ecriture: 29/05/93 (00:17:26)
  modification de l'etat: 29/05/93 (00:18:15)
$
```

Comme on peut le voir sur l'exemple qui suit, la commande **TOUCH(1)** réinitialise toutes les dates associées à un fichier :

```
$ date; touch TestDates ; lsdates TestDates
Mon Jul 19 17:48:09 MET DST 1993
  derniere lecture/ecriture: 19/07/93 (17:48:20)
  derniere ecriture: 19/07/93 (17:48:20)
  modification de l'etat: 19/07/93 (17:48:20)
$
```

Voici maintenant l'exemple de la commande `lstaille` affichant la taille de chacun des fichiers spécifiés dans la liste de ses arguments. Par défaut, elle imprime la taille de chaque fichier suivie de son nom. L'option `-s` permet de supprimer l'affichage du nom. Voici quelques exemples d'exécution de cette commande :

```
$ echo b*.[ch]
basename.c basename.h booleen.c booleen.h
$ lstaille b*.[ch]
     471 basename.c
     126 basename.h
      64 booleen.c
     129 booleen.h
$ lstaille b*.[ch] -s
     471
     126
      64
     129
$
```

Le source de cette commande est le suivant :

```
lstaille.c
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stddef.h>

#include "concatener.h"
#include "options.h"

#define MODE_SILENCIEUX   's'
#define LONGUEUR_CHAMP    8

char liste_options[] = { MODE_SILENCIEUX, '\0' };
main(int argc, char **argv)
{
    char *argv0 = argv[0];
    char *options = extraire_options(argv+1, liste_options);
    int sansnom;

    if (options == NULL)
    {
        fprintf(stderr, "Usage: %s -%s fichier...\n",
                argv0, liste_options);
        exit(1);
    }
    sansnom = options[0] == MODE_SILENCIEUX;

    while (++argv, --argc)
    {
        struct stat statbuf;
        char *pmessage = concatener(argv0, ":", *argv, NULL);

        if (**argv == '-')
            continue;
```

```
        if (stat(*argv, &statbuf) == -1)
        {
            perror(pmessage);
            continue;
        }
        printf("%*d", LONGUEUR_CHAMP, statbuf.st_size);
        if (!sansnom)
            printf(" %s", *argv);
        printf("\n");
    }
}
```

lstaille.c

Type d'un fichier

Le champ `st_mode` de la structure `struct stat` contient le type du fichier : régulier, répertoire... et son mode : droits d'accès, *set uid bit*... Le type du fichier s'obtient en masquant la valeur v de `st_mode` avec la constante `S_IFMT` prédéfinie dans `<sys/stat.h>`.

Ce masquage annule dans v tous les bits décrivant les droits d'accès et laisse inchangés ceux codant le type. Le résultat est l'une des valeurs suivantes :

- `S_IFDIR` : le fichier est un répertoire;
- `S_IFCHR` : le fichier est un fichier spécial associé à un pilote d'entrées-sorties en mode caractère;
- `S_IFBLK` : le fichier est un fichier spécial associé à un pilote d'entrées-sorties en mode bloc;
- `S_IFREG` : le fichier est un fichier régulier.

Il existe d'autres types de fichiers selon les versions d'UNIX (`S_IFLNK` : lien symbolique, `S_IFSOCK`: *socket*, `S_IFIFO`: tube nommé...). Un premier exemple simple illustrant le mécanisme de masquage du champ `st_mode` pour déterminer le type d'un fichier, est celui de la fonction `isdir` testant si son paramètre est le chemin d'un répertoire :

isdir.c

```
#include <sys/types.h>
#include <sys/stat.h>

#include "isdir.h"

int isdir(char *nom)
{
    struct stat bufstat;

    if (stat(nom, &bufstat) == -1)
        return -1;
    return (bufstat.st_mode & S_IFMT) == S_IFDIR;
}
```

isdir.c

Un second exemple est celui de la commande `lsg` qui extrait les noms de

répertoires (option `-d`), ou les noms de fichiers réguliers (option `-f`), d'une liste de noms de fichiers reçue en argument.

```
lsg.c
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <string.h>

#include "concatener.h"

static void usage(char *);

main(int argc, char **argv)
{
    /* Variables booleennes initialisees a faux */
    int fichier = 0;
    int repertoire = 0;
    char *argv0 = argv[0];

    /* Recuperation de l'option (-d ou -f) */
    if (argc == 1)
        usage(argv0);
    ++argv, --argc;
    if(strcmp(*argv, "-d") == 0)
        repertoire = 1;
    else if(strcmp(*argv, "-f") == 0)
        fichier = 1;
    else
        usage(argv0);

    /* Parcours des arguments */
    if (argc == 1)
        usage(argv0);
    while (++argv, --argc)
    {
        int type_de_fichier;
        struct stat bufstat;
        char *pmessage = concatener(argv0, ":", *argv, NULL);

        if (stat(*argv, &bufstat) == -1)
        {
            perror(pmessage);
            continue;
        }
        type_de_fichier = bufstat.st_mode & S_IFMT;
        if (fichier && type_de_fichier==S_IFREG
            || repertoire && type_de_fichier==S_IFDIR)
            printf("%s ", *argv);
    }
    printf("\n");
}

static void usage(char *s)
{
    fprintf(stderr, "Usage: %s -d|f fichier...\n", s);
    exit(1);
```

```
}
```

lsg.c

La première partie de la commande est classique : parcours des arguments et traitement des options; la commande teste ensuite le type des fichiers dont elle a reçu la liste en argument. La variable `type_de_fichier` reçoit le type du fichier. Le nom du fichier est affiché sur la sortie standard, si le fichier est un fichier régulier (`t==S_IFREG`) et la commande a été invoquée avec l'option `-f`, ou si le fichier est un répertoire (`t==S_IFDIR`) et la commande a été invoquée avec l'option `-d`.

La session suivante montre l'utilisation de cette commande sur un répertoire contenant trois fichiers (deux exécutables et un fichier archive) et deux répertoires. La commande UNIX **FILE(1)** liste le type de chacun des fichiers dont elle reçoit le nom en argument. Le caractère ? est le prompt affiché par le shell lorsque la saisie de la commande courante n'est pas terminée.

```
$ {
? echo
? file ge*
? echo
? echo - Fichiers:
? lsg -f ge*
? echo
? echo - Repertoires:
? lsg -d ge*
? }

gcc-cc1:        mc68k executable
gcc-cpp:        mc68k executable
gcc-include:    directory
gnu:            directory
libreadline.a:  5.2 archive

- Fichiers:
gcc-cc1 gcc-cpp libreadline.a

- Repertoires:
gcc-include gnu
$
```

On notera également le traitement des différents cas d'erreur:

```
$ lsg ding
Usage: lsg -d|f fichier...
$ lsg -g ding
Usage: lsg -d|f fichier...
$ lsg -f
Usage: lsg -d|f fichier...
$ lsg -f ding
lsg: ding inaccessible
$
```

Mode d'un fichier

Le mode d'un fichier décrit la liste des droits d'accès en lecture, en écriture et en exécution. Ces permissions sont définies pour trois classes d'utilisateurs :

- le propriétaire du fichier,
- les utilisateurs du même groupe que le groupe propriétaire du fichier excepté le propriétaire lui-même,
- les utilisateurs ne rentrant pas dans les deux premières catégories.

L'ensemble de ces permissions est représenté par une liste de neuf booléens, exprimés par un nombre de trois chiffres codé en octal. Ce sont les neuf bits de poids faible du champ `st_mode`. Par exemple, la constante octale `0640`, qui s'écrit en binaire 110100000, code le mode :

- 110: *lecture/écriture* pour le propriétaire
- 100: *lecture* pour le groupe propriétaire
- 000: aucun accès pour les autres

La notation UNIX de ce mode est `rw-r-----`. Elle se compose de trois groupes de trois caractères, le premier codant les accès du propriétaire, le second ceux du groupe, et le troisième ceux des autres utilisateurs. Chaque groupe est composé des caractères `r` ou `-`, `w` ou `-`, et `x` ou `-`.

Le mode d'un fichier code trois autres informations qui sont le *set uid* bit, le *set gid* bit et le *sticky* bit. Elles décrivent certaines caractéristiques concernant les fichiers exécutables (voir 12.1.1 page 411).

Le programme suivant affiche le mode d'accès d'un fichier selon la notation UNIX. Ce traitement est effectué par la fonction `mode` qui construit la chaîne de neuf caractères décrivant le mode d'un fichier dont elle reçoit le chemin en paramètre. Elle initialise un vecteur avec la chaîne `rwxrwxrwx`, et pour chaque bit à zéro dans le codage binaire du mode, remplace le caractère correspondant par un `-`. Le *set uid* (resp. *set gid*) bit est visualisé par un `s` minuscule ou majuscule remplaçant le `x` ou le `-` du champ utilisateur (resp. du champ groupe).

mode.c

```
#include <sys/types.h>
#include <sys/stat.h>

int mode(char *, char[9]);

main(int argc, char **argv)
{
    char md[9];

    while (argv++, --argc)
        if (mode(*argv, md))
            printf("%s: %s\n", *argv, md);
}

int mode(char *nom, char md[9])
{
    struct stat bufstat;
    short m;
    char *p = md+8;
```

```
    int i = 9;

    if (stat(nom, &bufstat) == -1)
        return 0;
    strcpy(md, "rwxrwxrwx");
    m =  bufstat.st_mode & 0xfff;
    while (i--)
    {
        if((m&1) == 0)
            *p = '-';
        p--;
        m >>= 1;
    }
    /* Traitement des set-uid et set-gid bits */
    if (m & 0x02)
        md[5] = md[5]=='-' ? 'S' : 's';
    if (m & 0x04)
        md[2] = md[2]=='-' ? 'S' : 's';
    return 1;
}
```
mode.c

Voici un exemple d'exécution de cette commande :

```
$ ls -l mode.c
-rw-r--r--  1 achille      737 May 29 00:30 mode.c
$
$ mode mode.c
mode.c: rw-r--r--
$
$ mode mode
mode: rwxr-xr-x
$
$ mode /bin/passwd
/bin/passwd: rwsr-xr-x
$
$ mode /usr/spool/mail/root
/usr/spool/mail/root: rw-------
$
```

On peut voir sur cet exemple que la commande **PASSWD(1)**, utilisée pour modifier les mots de passe, a le *set-uid bit* positionné.

11.1.3 Organisation des répertoires

Chemin d'un fichier

Le système de fichiers UNIX est organisé en arbre. Les fichiers réguliers et les fichiers spéciaux sont nécessairement des feuilles de cet arbre, c'est-à-dire des sommets ne pouvant pas avoir de descendant. Les répertoires sont des sommets susceptibles d'avoir des descendants.

Il n'existe pas de différence fondamentale entre un fichier régulier et un répertoire; tous deux sont identifiés par un inode décrivant leur état et donnant

accès à leurs données. Les données d'un répertoire sont des couples

$$\prec \textit{nom de fichier}, \textit{numéro d'inode} \succ$$

Chaque couple est appelé une **entrée**. Le format d'une entrée est décrit dans le fichier en-tête `<dir.h>`.

Le **chemin** d'un fichier est une liste de noms de répertoire terminée par un nom de fichier. Il décrit une liste formée alternativement :

- d'un inode de répertoire, pointant sur ses données, c'est-à-dire une liste d'entrées;
- d'une entrée de répertoire pointant sur un nouvel inode.

La figure 11.2 montre l'exemple du chemin `/bin/ls`.

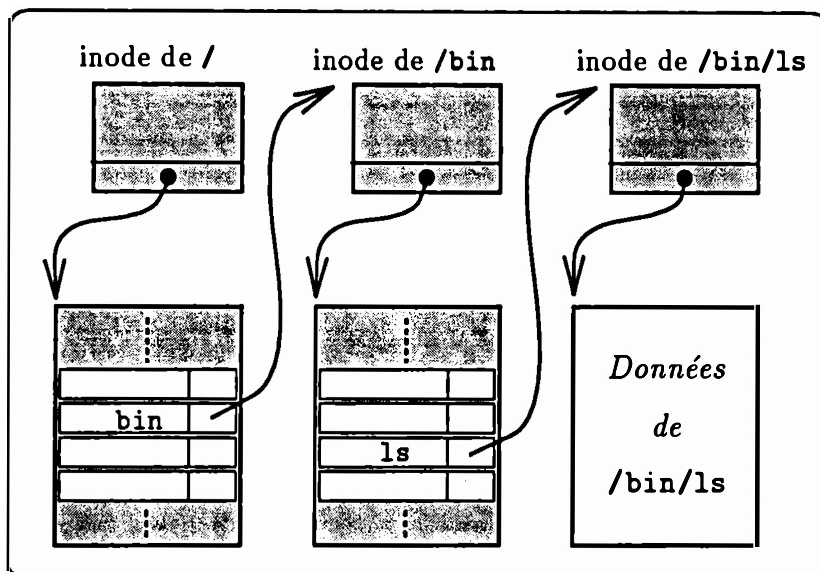


Figure 11.2: Chemin du fichier `/bin/ls`

Remarque 36 *Le chemin* `/bin/ls` *est un chemin absolu; le premier / représente la racine du système de fichiers. Un chemin d'accès est relatif si son premier caractère est différent de /. Dans ce cas, la recherche du fichier qu'il décrit s'effectue à partir du répertoire courant du processus qui effectue l'accès.*

Format d'une entrée de répertoire

`<sys/dir.h> <sys/types.h>`

Sous BSD, une entrée de répertoire est décrite par la structure `struct dirent` contenue dans le fichier `<sys/dir.h>`. Les champs 2, 3 et 4 de cette structure, `d_fileno`, `d_reclen` et `d_namlen` contiennent respectivement un numéro d'inode, la longueur totale de cette entrée et la longueur du nom du fichier associé à cet inode.

```
#define MAXNAMLEN    255

struct dirent
{
    off_t   d_off;
    u_long  d_fileno;
    u_short d_reclen;
    u_short d_namlen;
    char    d_name[MAXNAMLEN+1];
}
```

Le cinquième champ `d_name` est un tampon de la taille maximale d'un nom de fichier, `MAXNAMLEN`, augmentée de un pour contenir le marqueur de fin de chaîne, le caractère `NULL`.

Le parcours des données du répertoire s'effectue de la manière suivante :

1) lecture de la partie fixe d'une entrée de répertoire; sa taille est donnée par la constante `HEADSIZE` définie de la façon suivante :
   `#define HEADSIZE   (sizeof(struct dirent) - (MAXNAMLEN+1))`

2) lecture du nom du fichier, dont la longueur est donnée par le champ `d_namlen` de l'en-tête lu à l'étape 1;

3) défilement au moyen de l'appel système `lseek` de la fin de l'enregistrement.

En général, la fin de l'enregistrement contient quelques octets supplémentaires, dont le rôle est de réaligner le début l'enregistrement suivant sur une frontière de mot. Dans certains cas, en particulier à la fin du fichier, sa taille peut être supérieure à celle de la structure `struct dirent`; pour cette raison, il n'est pas possible de lire la partie variable directement par

```
read(fd, &dirent, dirent.d_reclen-HEADSIZE)
```

car la lecture s'effectuerait alors au-delà des limites de la structure. Le champ `d_off` de la structure `dirent` contient le décalage entre le début de l'enregistrement et le début du suivant; il tient compte des éventuels octets de réalignement.

Sous les anciennes versions de SYSTEM V, les entrées de répertoires sont de taille fixe. Cela présente un inconvénient majeur : la longueur des noms de fichiers est limitée. Une entrée de répertoire est décrite par la structure

```
struct dirent
{
    ino_t d_ino;
    char d_name[DIRSIZ];
};
```

où la pseudo-constante `DIRSIZ` est généralement définie par

```
#define DIRSIZ     14
```

Le parcours des données d'un répertoire sur lequel le descripteur `fd` est ouvert en lecture s'effectue de la façon suivante :

```
for (;;)
{
    struct dirent dir;

    if (read(fd, &dir, sizeof dir) < sizeof dir)
        break;
    if (dir.d_ino == 0)
        continue;
    faire quelque chose avec dir.d_name
}
```

Fonctions de lectures d'un répertoire

`<sys/dir.h> <sys/types.h>>`

Il existe un ensemble de fonctions, décrites dans la section **DIRECTORY(3)**, dont le but est de rendre le listage d'un répertoire indépendant de la version du système. Ces fonctions existent en général sous BSD et SYSTEM V. Ce sont :

- `opendir` : ouverture d'un flot en lecture sur un répertoire avec positionnement au début;
- `seekdir` : positionnement sur la i$^{\text{ème}}$ entrée,;
- `readdir` : lecture séquentielle d'une entrée à partir de la position courante;
- `closedir` : fermeture du flot.

La fonction `opendir` retourne un pointeur de type `DIR *`, utilisé comme paramètre des autres fonctions. Ces fonctions sont parfois absentes sous SYSTEM V. Elles peuvent dans ce cas être récrites sans grande difficulté, selon le principe évoqué ci-dessus.

La commande `lsdir` de l'exemple suivant parcourt le contenu d'une liste de répertoires et affiche, pour chacun, le nom de tous les fichiers qu'il contient, suffixé par un / lorsqu'il s'agit d'un répertoire.

```
========== lsdir.c ==========
#include <sys/types.h>
#include <sys/dir.h>
#include <stdio.h>

#include "isdir.h"
#include "concatener.h"

#define COLMAX 60

static char *argv0;
static void lsdir(char *);

main(int argc, char *argv[])
{
    argv0 = argv[0];
    if (argc == 1)
        lsdir(".");
    else
    {
        for (argv++; *argv!=NULL; argv++)
            lsdir(*argv);
```

```
    }
}


static void lsdir(char *nom)
{
    DIR *fdrep;
    int n = 0;
    int colmax = COLMAX;
    char *pmessage = concatener(argv0, ":", nom, NULL);

    printf("\n**%s** \n", nom);

    if (isdir(nom) != 1)
    {
        perror(pmessage);
        return;
    }

    fdrep = opendir(nom);
    if (fdrep == 0)
    {
        perror(pmessage);
        return;
    }
    for (;;)
    {
        struct direct *entree = readdir(fdrep);

        if (entree == 0)
            break;

        n += printf("%16s%c", entree->d_name,
                    (isdir(entree->d_name)== 1) ? '/' : ' ');
        if (n >= colmax)
        {
            n = 0;
            printf("\n");
        }
    }
    printf("\n");
}
```
lsdir.c

Dans l'exemple suivant elle est appelée avec trois chemins : le premier étant celui d'un répertoire d'accès interdit, le second celui d'un répertoire accessible, et le troisième un chemin invalide.

```
$ lsdir /export/root /tmp aabbcc

**/export/root**
lsdir:/export/root: Permission denied

**/tmp**
              ./             ../           .getwd  srv2512-firmin
 srv2516-firmin  #xmail18594#     12345678.tmp  pg-hc11.readme
```

```
    .getwda06577  .Xautha10934    .Xautha11091    .Xautha05797
    .Xautha11579   .FL4Q8z0K10     .FLC41A6ed0     .FLX0m21qU6
     .FL208uGan6   .FLKvYQAK10     .FL:4Ua6mh1     .FLz0jf0K10
     .FLTIkmmQK4   .FLbQkf3an6   Xternal.21378         Rs27841
     .FL:sFg1Wm8

**aabbcc**
lsdir:aabbcc: No such file or directory
$
```

On peut remarquer que les noms des fichier ne sont pas triés; le tri des noms est effectué par la commande `ls`, ou par l'interprète de commandes lors des expansions de noms de fichier. Il en est de même du masquage des fichiers dont le nom commence par un point. Les deux premières entrées sont attribuées aux noms de fichier "." et "..", le premier étant associé au répertoire lui-même et le second à son père.

Contrôles d'accès

<sys/file.h> ou **<unistd.h>**

Un processus ne peut accéder à un fichier que s'il a la permission de traverser les répertoires de son chemin d'accès. Les principales règles régissant les accès sont les suivantes :

- possibilité de lire un fichier si accès en lecture;
- possibilité d'écrire dans un fichier régulier ou spécial si accès en écriture;
- possibilité d'exécuter un fichier régulier (langage de commande ou exécutable pur) si accès en exécution;
- possibilité de créer ou de détruire un fichier si accès en écriture au répertoire le contenant;
- possibilité de traverser un répertoire, pour accéder à un de ses fichiers si accès en exécution.

Il n'est pas possible d'écrire directement dans un répertoire.

Les droits d'accès à un fichier d'un processus sont déterminés par :

- le champ *propriétaire* du mode, si le processus et le fichier ont même propriétaire;
- sinon, par le champ *groupe* du mode, s'ils sont du même groupe;
- sinon par le champ *autre* du mode.

Cet accès peut être testé au moyen de l'appel système **ACCESS(2)**. Il s'utilise avec deux paramètres, une chaîne de caractères contenant le chemin à analyser et une constante définissant l'accès à tester. Les constantes nécessaires à sa mise en œuvre sont définies dans le fichier en-tête **<sys/file.h>**, ou **<unistd.h>**; ce sont :

- `R_OK` : test de l'accès en lecture;
- `W_OK` : test de l'accès en écriture;
- `X_OK` : test de l'accès en exécution;
- `F_OK` : test de l'existence.

Les trois premières constantes sont combinables au moyen de l'opérateur |. Dans chaque cas, le système parcourt le chemin fourni en argument et vérifie que le processus qui a invoqué l'appel a la permission de traverser chacun des répertoires rencontrés. L'appel retourne la valeur zéro si le test d'accès est positif, et la valeur −1 sinon. Par exemple, à la suite de l'instruction suivante :

```
ok = access(NomFichier, R_OK|W_OK) == 0;
```

la variable `ok` vaut *vrai* si le fichièr dont le chemin est `NomFichier` est accessible en lecture et écriture, et *faux* sinon. Sur certains systèmes, les définitions des pseudo-constantes sont absentes. Il suffit de les redéfinir de la façon suivante :

```
#define R_OK 4
#define W_OK 2
#define X_OK 1
#define F_OK 0
```

11.1.4 Modification des attributs d'un fichier

Les attributs de fichier explicitement modifiables par un appel système sont le mode, le propriétaire, le groupe et les dates de modifications. Le nombre de liens, la taille, et les dates de modifications peuvent être modifiés implicitement par divers autres appels (création de liens, écriture...). Le numéro d'inode, le type et le numéro de *device* sont inaltérables. Ils sont fixés lors de la création du fichier, et demeurent inchangés jusqu'à sa destruction.

Modification du mode

Le mode d'un fichier est modifiable au moyen de l'appel **CHMOD (2)** qui s'utilise avec deux paramètres, le chemin du fichier et le nouveau mode d'accès codé par les douze bits de poids faible d'un entier. Lorsque ce second paramètre est une constante, il est d'usage de l'exprimer en base 8. Par exemple, l'exécution de

```
chmod("Data", 0666);
```

attribue le mode `rw-rw-rw-` au fichier `Data`. Bien entendu, seuls le super-utilisateur et le propriétaire d'un fichier ont le droit d'en modifier le mode.

Modification du propriétaire et du groupe

Seul le super-utilisateur peut modifier le propriétaire ou le groupe d'un fichier. Cette modification est effectuée au moyen de l'appel **CHOWN (2)**.

Modification des dates

Il est possible de modifier les dates de dernier accès et de dernière modification (voir 11.1.2) au moyen de l'appel système **UTIMES (2)**. Ces deux informations sont données dans cet ordre sous la forme d'un vecteur de structures `struct timeval`. La troisième date, celle de modification des attributs n'est pas directement modifiable. Elle est automatiquement mise à jour lors de toute modification d'un attribut, en particulier par l'appel `utimes`. Voici une version très simplifiée de la commande **TOUCH (1)** qui met à jour les dates d'accès à un

fichier. Elle utilise une forme simplifiée de mise en œuvre de l'appel `utimes`, dans laquelle le paramètre de temps est le pointeur `NULL`, et le temps utilisé par l'appel est le temps interne courant.

```
═══════ toucher.c ═══════════════════════════
#include <stddef.h>

main(int argc, char **argv)
{
    while (++argv, --argc)
    {
        utimes(*argv, NULL);
    }
}
═══════════════════════════════ toucher.c ═══════
```

On peut vérifier le comportement de la commande `toucher` en utilisant la commande `lsdates` définie page 372

```
$ lsdates toucher.c
  derniere lecture/ecriture: 22/07/89 (06:32:18)
  derniere ecriture: 22/07/89 (06:31:43)
  modification de l'etat: 22/07/89 (06:31:43)
$
$ toucher toucher.c ; lsdates toucher.c
  derniere lecture/ecriture: 22/07/89 (06:37:50)
  derniere ecriture: 22/07/89 (06:37:50)
  modification de l'etat: 22/07/89 (06:37:50)
$
```

11.2 Création et suppression de fichiers

11.2.1 Masque de création de fichiers

La création d'un nouveau fichier se décompose en deux étapes :

1) allocation et initialisation d'un inode;
2) allocation et initialisation d'une entrée dans le répertoire où il est créé.

Au départ, aucun bloc de données n'est alloué. Ils le seront au fur et à mesure des opérations d'écriture.

Nous avons vu en 10.2.1 que deux appels système permettent de créer un nouveau fichier, l'appel `open` sous sa forme étendue, et l'appel `creat` qui en est un cas particulier. Dans chaque cas, l'appel reçoit en paramètre le mode d'accès du fichier à créer.

Le fichier ainsi créé peut avoir des accès plus restreints que ceux spécifiés par le paramètre de mode d'accès. Ainsi, la commande suivante devrait créer le fichier dont le nom lui est passé en argument en mode lecture et écriture pour tous les utilisateurs (`rw-rw-rw-`).

```
═══════ creat.c ═══════════════════════════
main(int argc, char **argv)
{
    while (argv++, --argc)
        close(creat(*argv, 0666));
}
═══════════════════════════ creat.c ═══════
```

Comme on peut le voir, cela n'est pas le cas (on utilise la commande `mode` définie page 378 pour afficher le mode du fichier créé).

```
$ creat F1; mode F1
F1: rw-r--r--
$
```

En fait, l'argument de mode des appels `creat` et `open` est masqué avec un registre contenu dans l'environnement système du processus et hérité de son processus père. Ce masque, appelé `umask`, est utilisé de la façon suivante : chaque bit à un dans `umask` est forcé à zéro dans le mode d'accès du fichier créé. Pour le rendre inopérant, il suffit donc de lui donner la valeur zéro.

Il est possible de modifier la valeur du registre `umask` au moyen de l'appel du même nom. Voici une seconde version du programme précédent :

```
═══════ ucreat.c ═══════════════════════════
main(int argc, char **argv)
{
    umask(0);
    while (argv++, --argc)
        close(creat(*argv, 0666));
}
═══════════════════════════ ucreat.c ═══════
```

On remarquera que le paramètre de mode n'est utilisé que lorsque le fichier est réellement créé. S'il existe déjà au moment de l'appel, ses données sont réinitialisées mais son mode demeure inchangé.

```
$ ucreat F1 F2; mode F*
F1: -rw-r--r--
F2: -rw-rw-rw-
$
```

11.2.2 Verrouillage de ressources

Parmi les causes d'échecs de l'appel `creat` (ou `open` en option création), il en est une qui peut être exploitée de façon très intéressante. Il s'agit de la tentative de création d'un fichier déjà existant, et n'ayant pas d'accès en écriture. Considérons la situation suivante : deux processus ont exactement les mêmes accès et veulent pouvoir s'interdire temporairement l'un à l'autre l'accès à une ressource. Ce problème d'exclusion mutuelle est un problème classique de la programmation système. Les deux processus s'exécutant en temps partagé sur

un même processeur, le risque d'erreur est le suivant :

1) le premier processus P_1 teste l'accès à la ressource, obtient un résultat positif et est "endormi" par le système d'exploitation;
2) le second processus P_2 teste l'accès à la ressource, obtient un résultat positif, verrouille la ressource, commence à accéder à la ressource critique et est à son tour endormi par le système.;
3) le processus P_1 est réveillé, verrouille la ressource et y accède en même temps que P_2.

Un solution simple consiste à utiliser un fichier vide comme verrou de cette ressource. Tout accès à la ressource critique devra alors respecter le protocole suivant :

1) Tentative de création du fichier `verrou` en mode *"aucun accès"*.
2) Si succès, accès à la ressource, sinon attente (ou abandon).
3) Destruction du fichier `verrou`.

La création du fichier `verrou` s'effectuera de la façon suivante :

```
if (creat(VERROU, 0000) != -1)
{
    ...
    accès à la ressource
    ...
}
else
    ...
```

L'appel à `creat` s'exécute de la façon suivante :

```
si (le fichier verrou existe)
{
    si pas d'accès en écriture
        retour erreur
}
création du fichier verrou avec accès ---------
```

L'exclusion mutuelle est assurée par le système qui garantit un déroulement correct de l'appel `creat`. Les fonctions `verrouiller` et `deverrouiller` s'écrivent :

```
═══════ verrou.c ═══════════════════════════════
#include "verrou.h"

/* Retourne vrai si le verrouillage est reussi */
int verrouiller(char *nom)
{
    return creat(nom, 0) != -1;
}

/* Liberation de la ressource */
void deverrouiller(char *nom)
{
    unlink(nom);
}
═══════════════════════════════ verrou.c ═══════
```

La suppression du fichier est réalisée au moyen de l'appel **UNLINK(2)** que nous présentons dans la section suivante. Le nom du fichier verrou est passé en paramètre.

Voici un exemple testant le bon fonctionnement du verrouillage. Il s'agit d'une commande qui teste l'accès à la ressource en s'endormant pendant une seconde tant que la ressource est verrouillée. Lorsqu'elle accède à la ressource, elle simule un traitement en s'endormant durant cinq secondes, puis elle libère la ressource. Elle utilise la fonction `hms` présentée page 238 pour afficher l'heure courante. Il suffit de lancer en parallèle deux occurrences de ce programme pour tester le comportement lors d'accès concurrents à une même ressource. Pour pouvoir identifier l'auteur de chaque message affiché, ceux-ci sont préfixés par le numéro du processus qui en est l'auteur. Ce numéro est obtenu au moyen de l'appel **GETPID(2)** (voir 12.1.1).

test_verrou.c

```
#include <stdio.h>
#include "hms.h"
#include "verrou.h"

#define VERROU "/tmp/testverrou"

main()
{
    int h, m, s;
    int pid = getpid();

    printf("\n");
    while (! verrouiller(VERROU))
    {
        /* Attente de la ressource: un essai chaque seconde */
        hms(&h, &m, &s);
        printf("[%5d] a %02d:%02d:%02d attente\n",
               pid, h, m, s);
        sleep(1);
    }

    /* La ressource est accessible: verrouillage */
    hms(&h, &m, &s);
    printf("[%5d] a %02d:%02d:%02d verrouillage 5 sec.\n",
           pid, h, m, s);
    sleep(5);

    /* fin du travail et liberation de la ressource */
    hms(&h, &m, &s);
    printf("[%5d] a %02d:%02d:%02d liberation\n",
           pid, h, m, s);
    deverrouiller(VERROU);
}
```

test_verrou.c

Dans l'exemple qui suit, deux processus sont lancés en détaché, sous l'interprète de commandes. Le premier processus porte le numéro 10783 et le second 10784. C'est ici le second processus qui réussit à verrouiller la ressource.

```
$ verrou & verrou &
```

```
[1] 10783
[2] 10784
$
[10783] a 12:13:52 attente ressource
[10784] a 12:13:52 verrouillage 5 sec.
[10783] a 12:13:53 attente ressource
[10783] a 12:13:54 attente ressource
[10783] a 12:13:55 attente ressource
[10783] a 12:13:56 attente ressource
[10784] a 12:13:57 liberation ressource
[10783] a 12:13:58 verrouillage 5 sec.
[10783] a 12:14:03 liberation ressource
[2]  + Done                  verrou
[1]  + Done                  verrou
$
```

Remarque 37 *Il est également possible de gérer ce mécanisme avec une ouverture en mode* `O_CREAT|O_EXCL` *(voir page 333).*

11.2.3 Créations et suppressions de liens

On appelle **lien** l'association d'un chemin et d'un inode. Si on considère le modèle des inodes et des répertoires présenté au début de ce chapitre, il apparaît qu'il est tout à fait possible qu'un même inode soit référencé par plusieurs chemins différents. La création de plusieurs **liens** sur un même fichier est une propriété intéressante du système UNIX. La figure 11.3 représente deux répertoires différents `r1` et `r2`, le premier contenant une entrée de nom `f1` et le second une entrée de nom `f2`, les deux entrées faisant référence à un même inode, et par conséquent à un même fichier.

L'appel système permettant de créer un nouveau lien sur un fichier existant déjà est l'appel **LINK(2)** :

`int link(char *`$\langle chemin_1 \rangle$`, char *`$\langle chemin_2 \rangle$`)`

Le paramètre $\langle chemin_1 \rangle$ est le chemin d'un fichier existant déjà, et le paramètre $\langle chemin_2 \rangle$ est le chemin du lien à créer. Pour des raisons de cohérence du système que nous ne détaillerons pas ici, seul le super-utilisateur est autorisé à créer un lien sur un répertoire. Une fois le lien créé, plus rien ne distingue le chemin original du nouveau chemin; les deux liens jouent un rôle parfaitement symétrique. Les principales raisons d'échec de l'appel `link` sont les suivantes :

- $\langle chemin_2 \rangle$, le lien à créer existe déjà;
- $\langle chemin_1 \rangle$, le fichier sur lequel porte le lien n'existe pas;
- le répertoire dans lequel doit être créé le lien n'est pas accessible en écriture.

La destruction d'un lien est effectuée par l'appel système **UNLINK(2)** :

`int unlink(char *`$\langle chemin \rangle$`)`

Pour détruire un lien, il est nécessaire d'avoir accès en écriture au répertoire le contenant.

Chaque inode contient un compteur du nombre de liens qui le référencent. Chaque appel à `unlink` décrémente ce compteur de références. La place occupée sur disque par l'inode et les données n'est libérée que lorsque le compteur de références est nul et qu'aucun processus ne possède d'entrée-sortie ouverte sur ce fichier.

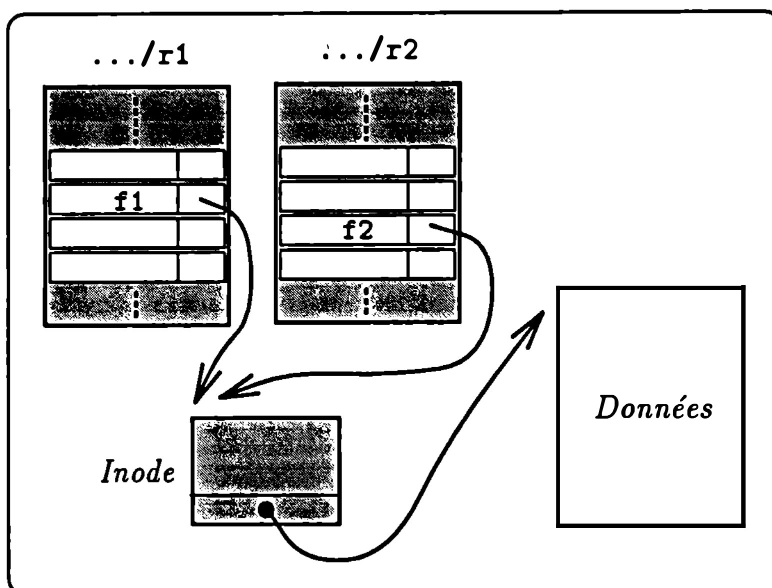


Figure 11.3: Deux liens sur un même fichier.

On peut mettre en évidence ce fonctionnement au moyen de la commande `retenir` qui ouvre un fichier en lecture et s'endort pendant un temps dont la valeur est passée en argument. Lorsqu'elle se réveille, elle termine son exécution, ce qui provoque la fermeture de l'entrée-sortie. Le source de cette commande est très simple à écrire :

```
retenir.c
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>

main(int argc, char *argv[])
{
    if (argc != 3)
    {
        fprintf(stderr,
                "Usage: %s <fichier> <secondes>\n", argv[0]);
        exit(1);
    }
    if (open(argv[1], O_RDONLY) != -1)
        sleep(atoi(argv[2]));
}
retenir.c
```

La session suivante enchaîne les étapes :

1) affichage de l'espace libre sur la partition physique de la racine du système

de fichier, soit 3330 blocs;

2) copie d'un fichier de 2272 blocs, le fichier

`/usr/new/splash`

dans le répertoire `/tmp`, et affichage de la nouvelle valeur de l'espace libre, soit 1058 blocs;

3) lancement en détaché de la commande :

`retenir /tmp/splash 15`

4) suppression du fichier au moyen de la commande `rm` et affichage des nouvelles caractéristiques de la partition; elles n'ont pas changé;

5) affichage des caractéristiques après la terminaison de `retenir` : la place occupée par `/tmp/splash` a été libérée.

```
$ df | head -2
Filesystem    kbytes    used   avail capacity  Mounted on
/dev/sd0a       7437    3363    3330    50%    /
$ ls -s /usr/new/splash
2272 /usr/new/splash
$
$ cp /usr/new/splash /tmp ; df | head -2
Filesystem    kbytes    used   avail capacity  Mounted on
/dev/sd0a       7437    5635    1058    84%    /
$ retenir /tmp/splash 15 &
[5] 28139
$ rm /tmp/splash ; df | head -2
Filesystem    kbytes    used   avail capacity  Mounted on
/dev/sd0a       7437    5635    1058    84%    /
$
[5]+  Done                 retenir /tmp/splash 15
$ df | head -2
Filesystem    kbytes    used   avail capacity  Mounted on
/dev/sd0a       7437    3363    3330    50%    /
$
```

Le renommage de fichier est un cas particulier de la création de liens. En effet, pour modifier le nom d'un fichier existant déjà, il suffit de :

1) créer un lien avec le nouveau nom sur le fichier référencé par l'ancien nom;
2) détruire l'ancien nom.

La commande `rename`, présentée sur l'exemple suivant, montre la mise en œuvre de ce mécanisme. On notera les différentes situations d'erreurs possibles lors du retour de l'appel à `unlink` :

- `EACCES` : l'accès à un des chemins est interdit;
- `EEXIST` : le second chemin existe déjà;
- `ELOOP` : le décodage d'un des chemins suit trop de liens symboliques;
- `EMLINK` : le nombre maximum de liens sur ce fichier est atteint;
- `ENOENT`, `ENOTDIR` : un des chemins est incorrect;
- `EPERM` : le premier argument est un répertoire et le processus n'appartient pas au super-utilisateur;

– EXDEV : les deux chemins aboutissent à des systèmes de fichiers différents.

rename.c

```
#include <sys/file.h>
#include <errno.h>
#include <stdio.h>

#include "concatener.h"

static void error(char *);

main(int argc, char *argv[])
{
    char *pmessage1;
    char *pmessage2;
    char *pmessage12;

    if (argc != 3)
    {
        fprintf(stderr, "Usage: rename oldname newname\n");
        exit(1);
    }

    pmessage1 = concatener(argv[0], ":", argv[1], NULL);
    pmessage2 = concatener(argv[0], ":", argv[2], NULL);
    pmessage12 = concatener(argv[0], ":", argv[1],
                            ",", argv[2], NULL);

    if (access(argv[1], F_OK) == -1)
        error(pmessage1);

    /* Creation du lien avec le nouveau nom */
    if (link(argv[1], argv[2]) == -1)
    {
        switch(errno)
        {
          case EMLINK:
          case EPERM:
            error(pmessage1);

          case EEXIST:
            error(pmessage2);

          case EACCES:
          case ELOOP:
          case ENOENT:
          case ENOTDIR:
          case EXDEV:
            error(pmessage12);

          default:
            error(argv[0]);
        }
    }
    /* Suppression de l'ancien nom */
    if (unlink(argv[1]) == -1)
        error(pmessage1);
```

```
}

static void error(char *message)
{
    perror(message);
    exit(1);
}
```
rename.c

Le motif F? utilisé en argument de la commande `ls` désigne tous les fichiers du répertoire courant dont le nom commence par la lettre F suivie de n'importe quel caractère.

```
$ ls F?
F1
$ cat F1
Contenu de F1
$
$ rename F1 F2; ls F?
F2
$ cat F2
Contenu de F1
$ cp F2 F1; ls F?
F1      F2
$
$ rename F2 F1
rename:F1: File exists
$ rename F3 F1
rename:F3: No such file or directory
$
$ rename . F2
rename:.: Not owner
$
```

Cette implémentation ne permet pas de renommer les répertoires, puisqu'il n'est pas permis à un utilisateur normal d'effectuer des liens dessus. Sur de nombreuses versions d'UNIX, il existe un appel **RENAME(2)** permettant de renommer fichiers et répertoires.

La suppression de fichier régulier ou de fichier spécial ne pose pas de problème particulier. Voici un exemple de programme utilisant plusieurs appels système présentés dans ce chapitre; il s'agit d'une commande de suppression de fichiers baptisée `sp`. Elle utilise les fonctions :

- `int isdir(char *)` : teste si son paramètre est un chemin de répertoire (voir page 374);
- `int dirname(char *)` : extrait, dans le chemin d'un fichier, celui du répertoire qui le contient (voir page 219);
- `int getok(char *)` : demande une confirmation de suppression pour le fichier dont le nom lui est passé en paramètre;
- `void erreur(char *, char *)` : affichage d'un message d'erreur sur la sortie d'erreur standard `stderr`.

La variable booléenne `interactif` est positionnée à *vrai* s'il faut demander

une confirmation de suppression. On remarquera les différents tests mis en œuvre pour vérifier la légalité de la suppression.

La fonction principale itère l'appel à la fonction **sp** sur chacun des noms de fichiers reçus en arguments. Elle détecte la présence éventuelle de l'option `-i`, demandant une confirmation interactive des suppressions, et positionne le cas échéant la variable booléenne globale **interactif**. Les fonctions d'entrées-sorties **erreur** et **getok** ne présentent pas de difficulté particulière.

sp.c

```
#include <unistd.h>

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

/* Nom de la commande (argv[0]) */
static char *cmd;

/* Environnement pour la fonction sp() */
static int    interactif = 0;

static void usage(char *s);
static void sp(char *nom);
static void erreur(char *f, char *n);
static int getok(char *s);

main(int argc, char **argv)
{
    /* Recuperation des arguments */
    cmd = *argv++;
    argc--;
    if (**argv == '-')
    {
        if (strcmp(*argv, "-i") == 0)
            interactif = 1;
        else
            usage(cmd);
        argv++, argc--;
    }
    /* Il doit y avoir au moins un nom */
    if (argc == 0)
        usage(cmd);

    /* Traitement de la liste de noms */
    do
    {
        sp(*argv);
    }
    while (argv++, --argc);
}

static void usage(char *s)
{
    fprintf(stderr, "Usage: %s [-i] nom... \n", s);
    exit(1);
}
```

```
/* Suppression du fichier 'nom' */
static void sp(char *nom)
{
    /* Test d'existence du fichier */
    if (access(nom, F_OK) == -1)
    {
        erreur("impossible d'acceder a \"%s\"\n", nom);
        return;
    }
    /* Acces en ecriture au repertoire pere */
    if (access(dirname(nom), W_OK) == -1)
    {
        erreur("suppression de \"%s\" interdite\n", nom);
        return;
    }
    /* Erreur si repertoire */
    if (isdir(nom) == 1)
    {
        erreur("\"%s\" est un repertoire\n", nom);
        return;
    }
    /* Demande de confirmation en mode interactif */
    if (interactif && !getok(nom))
        return;
    /* Destruction du fichier */
    if (unlink(nom) == -1)
    {
        erreur("suppression de \"%s\" impossible\n", nom);
        return;
    }
}

static void erreur(char *f, char *n)
{
    fprintf(stderr, "%s: ", cmd);
    fprintf(stderr, f, n);
}

static int getok(char *s)
{
    char buffer[8];

    printf("?suppression de \"%s\" (o/n): ", s);
    fflush(stdout);
    if (read(0, buffer, sizeof buffer) <= 0)
        exit(0);
    if (*buffer == 'q')
        exit(0);
    return *buffer == 'o' || *buffer == 'O';
}
```

sp.c

La session qui suit montre les différentes situations pouvant se présenter lors de l'exécution de cette commande. L'exemple se décompose en deux parties :

- création d'une douzaine de fichiers au moyen d'une boucle effectuée sous l'interprète de commandes **sh**;

– exécutions successives de la commande **sp** illustrant les différentes situations possibles.

```
$ for N in a b c d; do
?   for I in 1 2 3; do
?     > $N$I.f
?   done
? done
$ mkdir a4.r b3.r b2.r ; ls -CF
a1.f    a3.f    b1.f    b2.r/   b3.r/   c2.f    d1.f    d3.f
a2.f    a4.r/   b2.f    b3.f    c1.f    c3.f    d2.f
$
$ sp -i a*
?suppression de "a1.f" (o/n): o
?suppression de "a2.f" (o/n): n
?suppression de "a3.f" (o/n): o
sp: "a4.r" est un repertoire
$
$ sp b*
sp: "b2.r" est un repertoire
sp: "b3.r" est un repertoire
$ ls -CF
a2.f    a4.r/   b2.r/   b3.r/   c1.f
c2.f    c3.f    d1.f    d2.f    d3.f
$
$ chmod u-w . ; sp b3.r ; chmod u+w .
sp: suppression de "b3.r" interdite
$ sp gondor
sp: impossible d'acceder a "gondor"
$
```

11.2.4 Répertoires et fichiers spéciaux

Il est possible de créer directement un inode au moyen de l'appel **MKNOD(2)**. L'appel reçoit en paramètres le chemin du lien référençant cet inode, et son mode (accès et type). C'est de cette façon que l'on crée un fichier spécial. Cet appel ne peut être utilisé par un utilisateur normal que pour créer des tubes nommés, qui sont des unités d'entrées-sorties se comportant comme les tubes anonymes, mais référencées par un chemin comme un fichier. Ainsi, l'appel

```
mknod(chemin, S_IFIFO | 0666)
```

crée un tube nommé dont le nom est `chemin` et qui est accessible en lecture et écriture à tout utilisateur.

Bien que leur structure soit identique à celle des fichiers réguliers, la création et la suppression des répertoires sont soumises à certaines restrictions. En particulier, il n'est pas possible de détruire directement un répertoire qui contient des fichiers, sous peine de perdre la place occupée sur disque par ces fichiers. Il faut donc détruire un par un tous les fichiers qu'il référence; si certains de ces fichiers sont eux-mêmes des répertoires, il est alors nécessaire d'enchaîner

récursivement la suppression sur chacun d'entre eux.

De même, la création d'un nouveau répertoire n'est pas une opération élémentaire, car elle comprend la construction des entrées "." et "..", référençant respectivement le répertoire courant et le répertoire père. D'autres problèmes se pausent, beaucoup plus complexes, que nous ne développerons pas ici, pouvant entraîner des blocages au niveau du système lui-même.

Sous BSD et sous certaines versions de SYSTEM V, les deux appels système **MKDIR(2)** et **RMDIR(2)** effectuent respectivement la création et la suppression d'un répertoire. Il est de cette façon possible à un utilisateur de manipuler des répertoires depuis un programme.

Voici par exemple le source de la fonction `spr` effectuant une suppression récursive à partir d'un chemin fourni en paramètre. Pour parcourir le contenu d'un répertoire, on utilise les fonctions de la bibliothèque **DIRECTORY(3)** (voir page 381) permettant de parcourir les entrées d'un répertoire. L'algorithme mis en œuvre dans la fonction `spr` est le suivant :

```
spr(nom):
    si (nom est un répertoire)
        descente dans le répertoire nom
        pour toute entrée n de nom
            spr(n)
        remontée dans le répertoire père
        destruction du répertoire nom
    sinon
        destruction du fichier nom
```

Le déplacement dans l'arborescence s'effectue en changeant le répertoire courant du processus, au moyen de l'appel système **CHDIR(2)** (voir 12.1.1). Lorsque la variable `bavard` est différente de zéro, la fonction `sp` imprime une trace de toutes les suppressions effectuées; elle retourne 0 si une erreur survient lors du traitement et 1 sinon. La fonction principale de la commande `spr` itère l'appel à `sp` sur chacun de ses arguments. Si l'option `-v` est présente, elle positionne la variable `bavard` à 1.

```
spr.c
#include <sys/types.h>
#include <sys/dir.h>
#include <sys/stat.h>
#include <stdio.h>

static int bavard = 0;
int spr(char *);

main(int argc, char **argv)
{
    char *argv0 = argv[0];

    if (strcmp(argv[1], "-v") == 0)
        bavard = 1, argv++, argc--;

    while (argv++, --argc)
        if (!spr(*argv))
            fprintf(stderr, "%s: impossible de detruire \"%s\"\n",
                    *argv0, *argv);
```

```
}

int spr(char *nom)
{
    struct stat bufstat;

    if (stat(nom, &bufstat) == -1)
        return 0;
    if ((bufstat.st_mode&S_IFMT) == S_IFDIR)
    {
        /* 'nom' est un repertoire */
        DIR *fdrep = opendir(nom);

        if (fdrep == 0)
            return 0;
        /* Descente dans le repertoire */
        if (chdir(nom) == -1)
            return 0;
        /* On defile les entrees "." et ".." */
        seekdir(fdrep, 2);
        for (;;)
        {
            struct direct *entree = readdir(fdrep);

            if (entree == 0)
                break;
            /* appel recursif sur chaque fichier du rep. */
            if (spr(entree->d_name) == 0)
                return 0;
        }
        /* remonte dans le repertoire pere */
        closedir(fdrep);
        chdir("..");
        if (rmdir(nom) == -1)
            return 0;
        if (bavard)
            printf("%s/ detruit\n", nom);
    }
    /* si nom est un fichier */
    else
    {
        if (unlink(nom) == -1)
            return 0;
        if (bavard)
            printf("%s detruit\n", nom);
    }
    return 1;
}
```

spr.c

L'exemple suivant montre une exécution de la commande `spr` sur une arborescence de racine `t1`, qu'on visualise auparavant en faisant exécuter la commande `ls -RF` parcourant récursivement une arborescence.

```
$ ls -RF t1
f11     t11/    t12/    t13/
t1/t11:
```

```
f111    f112
t1/t12:
t1/t13:
f131    f132
$
$ spr -v t1
t12/ detruit
f11 detruit
f131 detruit
f132 detruit
t13/ detruit
f111 detruit
f112 detruit
t11/ detruit
t1/ detruit
$
```

11.3 Gestion de fichiers indexés

Il est très simple grâce à l'appel `lseek` d'organiser un fichier UNIX en enregistrements de taille fixe, exploitables en accès direct. L'enregistrement du fichier est décrit au moyen d'une structure. Soit par exemple `struct` ⟨*enr*⟩ le type d'un enregistrement et ⟨*enr*⟩ un objet de ce type. Le positionnement sur le n^{ème} enregistrement s'effectue par un décalage de $n - 1$ fois la taille de la structure:

`lseek(fd, sizeof(struct ⟨enr⟩)*(n-1), 0)`

et l'écriture de l'enregistrement se fait par :

`write(fd, &⟨enr⟩, sizeof ⟨enr⟩)`

La relecture s'effectue de façon symétrique : positionnement, puis

`read(fd, &⟨enr⟩, sizeof ⟨enr⟩)`

Pour illustrer ce mécanisme, nous allons traiter l'exemple d'un fichier de chaînes de caractères, structuré au moyen d'un fichier d'index. Plus précisément, nous avons :

- un fichier de chaînes de caractères de longueurs quelconques; les chaînes sont rangées les unes à la suite des autres, chacune étant terminée par le caractère `\0`;
- un fichier d'index constitué d'enregistrements de taille fixe, contenant pour chaque chaîne, la position de son premier caractère dans le fichier des chaînes et sa longueur.

Il est possible d'effectuer un accès direct sur n'importe quelle chaîne, la lecture de la $i^{ème}$ chaîne s'effectuant de la façon suivante :

- positionnement sur le $i^{ème}$ enregistrement du fichier d'index;
- lecture de cet index, et positionnement sur le début de la chaîne dans le fichier des chaînes;

– lecture des caractères de la chaîne.

La structure de ces deux fichiers est détaillée sur la figure 11.4.

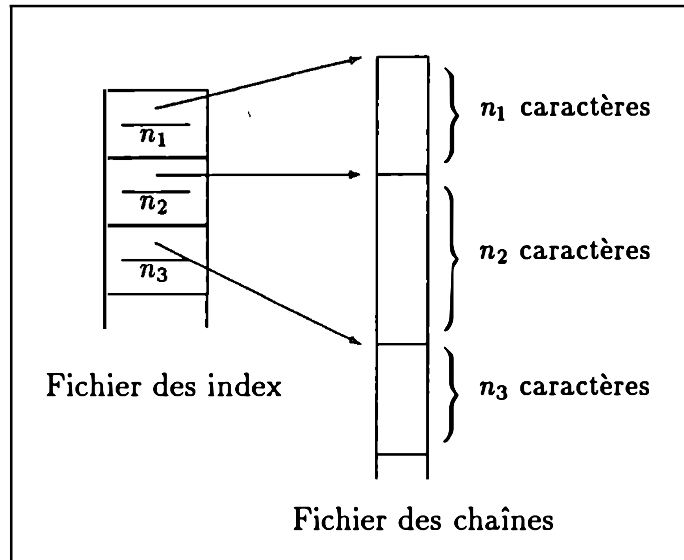


Figure 11.4: Organisation des fichiers d'index et de chaînes

Le fichier `accind.c` contient ce module de gestion d'accès indexé. Il est composé des quatre fonctions suivantes :

- `accind_ouvrir` : ouverture des fichiers index et chaîne en mode lecture-écriture; s'ils n'existent pas, les fichiers sont créés;
- `accind_fermer` : fermeture des deux fichiers;
- `accind_ajouter` : ajout d'une chaîne à la liste des chaînes déjà saisies;
- `accind_lire` : lecture de la n$^{\text{ème}}$ chaîne.

La fonction `accind_ouvrir` utilise une fonction auxiliaire locale au fichier. Le fichier d'index est formé d'enregistrements de taille fixe, dont la forme est donnée par la structure suivante :

```
struct index
{
    long debut;
    long longueur;
}
```

Les deux fichiers portent le même nom de base auquel est concaténé un suffixe propre. Les pseudo-constantes `SUFFIXE_INDEX` et `SUFFIXE_ENREG` définissent ces suffixes. Les variables `fd_enreg` et `fd_index`, déclarées locales au fichier `accind.c` contiennent les descripteurs associés aux deux fichiers.

```
#define SUFFIXE_INDEX ".ndx"
#define SUFFIXE_ENREG ".enr"

static int fd_index;
static int fd_enreg;
```

```
    }
    return 1;
}


static int accind_ouvriraux(char *prefixe, char *suffixe)
{
    char *nom = malloc(strlen(prefixe) + strlen(suffixe) + 1);
    int fd;

    strcpy(nom, prefixe);
    strcat(nom, suffixe);
    fd = open(nom, O_RDWR|O_CREAT, 0644);
    free(nom);
    return fd;
}

void accind_fermer()
{
    close(fd_enreg);
    close(fd_index);
}

void accind_ajouter(char *enreg)
{
    struct index index;
    int longueur = strlen(enreg);

    index.debut = lseek(fd_enreg, 0L, 2);
    index.longueur = longueur;
    write(fd_enreg, enreg, longueur);
    lseek(fd_index, 0L, 2);
    write(fd_index, &index, sizeof index);
}

char *accind_nieme(int n)
{
    struct index index;
    long decalage =(n-1) * sizeof index;
    char *enrg;

    if (lseek(fd_index, decalage, 0) != decalage)
        return NULL;
    if (read(fd_index, &index, sizeof index) != sizeof index)
        return NULL;
    if (lseek(fd_enreg, index.debut, 0) != index.debut)
        return NULL;
    enrg = malloc(index.longueur+1);
    if (read(fd_enreg, enrg, index.longueur) != index.longueur)
    {
        free(enrg);
        return NULL;
    }
    enrg[index.longueur] = '\0';
    return enrg;
}
```

accind.c

La fonction `accind_ouvrir` reçoit en argument le nom de base des fichiers à ouvrir; la fonction auxiliaire `accind_ouvriraux` construit le nom respectif de chacun d'entre eux et effectue un appel à l'appel `open`.

L'ajout d'une nouvelle chaîne est effectué par la fonction

```
void accind_ajouter(char *enreg)
```

Il ne présente pas de difficulté particulière. Il suffit de placer l'index de position de chaque entrée-sortie en fin de fichier au moyen de l'appel `lseek`. La valeur retournée lors du positionnement dans le fichier des chaînes permet d'initialiser le champ `debut` de l'enregistrement d'index. La longueur est calculée au moyen de la fonction `strlen`.

La lecture de la n^{ème} chaîne est effectuée par la fonction

```
char *accind_nieme(int n)
```

Elle enchaîne les étapes suivantes :

– lecture du n^{ème} index, ce qui donne la position et la longueur de la chaîne;
– allocation d'une zone mémoire de la longueur de la chaîne;
– positionnement dans le fichier des chaînes;
– lecture de la chaîne dans la zone mémoire allouée dynamiquement.

La fonction est écrite de manière à retourner une chaîne vide lorsque le numéro de chaîne demandé est supérieur au nombre maximum de chaînes.

Voici le listing complet de ce module :

```
═══════════ accind.c ═══════════════════════════════════
#include <stddef.h>
#include <stdlib.h>
#include <fcntl.h>

#include "accind.h"

#define SUFFIXE_INDEX   ".ndx"
#define SUFFIXE_ENREG   ".enr"

struct index
{
    long debut;
    long longueur;
};

static int fd_index;
static int fd_enreg;

static int accind_ouvriraux(char *, char *);

int accind_ouvrir(char *prefixe)
{
    fd_enreg = accind_ouvriraux(prefixe, SUFFIXE_ENREG);
    if (fd_enreg == -1)
        return 0;
    fd_index = accind_ouvriraux(prefixe, SUFFIXE_INDEX);
    if (fd_index == -1)
    {
        close(fd_enreg);
        return 0;
```

Ce module peut être testé au moyen du programme de saisie suivant :

```
═══════ saisie.c ═══════════════════════════
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

#include "lire_chaine.h"
#include "accind.h"

static void usage(char *);

main(int argc, char *argv[])
{
    if (argc != 2)
        usage(argv[0]);
    if (!accind_ouvrir(argv[1]))
        exit(1);
    for (;;)
    {
        char *p = lire_chaine("-> ");

        if (*p == '\0')
            break;
        accind_ajouter(p);
        free(p);
    }
    accind_fermer();
}

static void usage(char *s)
{
    fprintf(stderr, "Usage: %s fichier\n", s);
    exit(1);
}
═══════════════════════════ saisie.c ═══════
```

Ce programme utilise la fonction `lire_chaine` définie page 222. La saisie s'arrête sur la lecture d'une chaîne vide. Il est possible d'enchaîner plusieurs saisies, comme on peut le voir sur l'exemple suivant :

```
$  saisie Phrase
-> Premiere phrase du fichier des phrases
-> la deuxieme phrase vient en seconde position
->
$ ls -l Phrase.*
-rw-r--r--  1 achille      101 Mar 18 14:14 Phrase.chn
-rw-r--r--  1 achille       16 Mar 18 14:14 Phrase.ndx
$ saisie Phrase
-> Derniere phrase (pour l'instant)
->
$ ls -l Phrase.*
-rw-r--r--  1 achille      146 Mar 18 14:14 Phrase.chn
-rw-r--r--  1 achille       24 Mar 18 14:14 Phrase.ndx
```

```
$
```

La taille d'une structure `index` est ici de huit octets. On peut vérifier que la taille du fichier des index est de $8 \times n$ octets, n étant le nombre de phrases saisies.

Le même module d'accès indexé peut être utilisé pour écrire une commande de lecture dans le fichier indexé. La commande suivante effectue la lecture de la chaîne dont le rang lui est passé en argument. Si le numéro est incorrect, la fonction `accind_nieme` retourne la valeur `NULL` et la commande n'imprime rien.

```
consult.c
#include <stdio.h>
#include <stddef.h>
#include <ctype.h>

#include "accind.h"

void usage(char *);

main(int argc, char *argv[])
{
    char *s;
    int n;

    if (argc != 3)
        usage(argv[0]);

    n = atoi(argv[2]);
    if (!accind_ouvrir(argv[1]))
        exit(1);
    s = accind_nieme(n);
    if (s != NULL)
    {
        printf("  Chaine numero %d: %s\n", n, s);
        free(s);
    }
    accind_fermer();
}

void usage(char *s)
{
    fprintf(stderr, "Usage: %s fichier numero-enreg.\n", s);
    exit(1);
}
consult.c
```

Voici quelques exemples de lectures au moyen de la commande `consult` de lignes saisies dans la session de l'exemple précédent :

```
$ consult Phrases 3
  Chaine numero 3: Derniere phrase (pour l'instant)
$ consult Phrases 6
$
```

11.4 Utilisation de fichiers système

Le répertoire /etc contient, sous UNIX, les différents fichiers de données utilisés par le système, comme par exemple le fichier passwd où sont définis les utilisateurs, ou sur BSD le fichier termcap décrivant les capacités vidéo de diverses consoles.

Certains de ces fichiers, comme les deux cités précédemment, sont exploitables au moyen de bibliothèques de fonctions. D'autres fichiers, pour lesquels cela n'est pas le cas, sont cependant décrits dans la documentation du système, comme par exemple le fichier utmp qui contient, à tout instant, la liste des utilisateurs connectés sur la machine.

Le fichier utmp est décrit en **UTMP(4)** sur SYSTEM V et en **UTMP(5)** sur BSD. La structure d'un enregistrement du fichier est définie dans le fichier entête <utmp.h>. Cette structure diffère malheureusement d'une version à l'autre du système. Certaines informations sont communes aux deux, comme le nom de l'utilisateur, ou celui de la ligne sur laquelle il est connecté (c'est-à-dire le nom du fichier spécial associé).

Nous allons voir un exemple de commande, la commande message, utilisant deux fichiers du système :

- le fichier passwd, exploité au moyen des fonctions de la bibliothèque **GETPWENT(3)**;
- le fichier utmp, lu enregistrement par enregistrement, en utilisant le type struct utmp défini dans <utmp.h>.

Les champs de la structure struct utmp utilisés pour l'écriture de message sont :

- ut_name (BSD) ou ut_user (SYSTEM V) : le nom de l'utilisateur;
- ut_line : le nom du fichier spécial pilotant le terminal attaché à la connexion.

La commande message reçoit en arguments un nom d'utilisateur et un message à lui transmettre s'il est connecté. Elle vérifie tout d'abord que cet utilisateur existe, c'est-à-dire qu'il est défini dans le fichier passwd, en utilisant pour cela les fonctions définies dans getpwent.

La bibliothèque getpwent est constituée de fonctions permettant de défiler le fichier /etc/passwd en séquentiel et de rechercher l'enregistrement contenant un nom ou un numéro d'utilisateur. Nous utilisons ici la fonction getpwnam qui reçoit en argument un nom d'utilisateur, et retourne sa description s'il existe et la valeur NULL sinon.

Si l'utilisateur existe, il est recherché séquentiellement dans le fichier utmp, en itérant la lecture d'une structure de type struct utmp. Dans le cas où il est connecté, il existe un enregistrement du fichier le décrivant. Le champ ut_line correspondant contient le nom du fichier spécial associé à sa ligne. La commande message tente d'ouvrir le terminal de l'utilisateur destinataire du message en écriture. Si l'ouverture réussit, il est alors possible d'y recopier le message.

Ce programme utilise la fonction **GETLOGIN(3)** qui retourne le nom de *login* correspondant à l'identificateur propriétaire du processus. On notera également les définitions de pseudo-constantes et pseudo-fonctions assurant la portabilité de cette commande sur les deux versions du système.

```
message.c
#define BSD  1
#define SYSV 0

#include <pwd.h>
#include <fcntl.h>
#if SYSV
#  include <sys/types.h>
#endif
#include <utmp.h>
struct utmp utmp;

#define LINESIZE   (sizeof utmp.ut_line)
#if BSD
#  define NAME(u)  ((u).ut_name)
#endif
#if SYSV
#  define NAME(u)  ((u).ut_user)
#endif

char nom_tty[LINESIZE] = { '/', 'd', 'e', 'v', '/', };
char *getlogin();

static void message(int, char *, char *);

main(int argc, char **argv)
{
    int fd;
    char *nom_ut = argv[1];

    if (argc < 3)
    {
        message(2, "Usage: %s <util> <message>\n", *argv);
        exit(0);
    }
    /* Verification de l'existence de l'utilisateur */
    if (getpwnam(nom_ut) == 0)
    {
        message(2, "%s n'existe pas\n", nom_ut);
        exit(1);
    }
    endpwent();

    /* Recherche du nom d'utilisateur dans /etc/utmp */
    fd = open("/etc/utmp", O_RDONLY);
    for (;;)
    {
        if (read(fd, &utmp, sizeof utmp) < sizeof utmp)
        {
            message(1, "%s n'est pas connecte\n", nom_ut);
            exit(0);
        }
```

```
            if (strncmp(NAME(utmp),nom_ut,sizeof NAME(utmp)) == 0)
            {
                strcat(nom_tty, utmp.ut_line);
                break;
            }
        }
        close(fd);

        /* Tentative d'ouverture du pilote tty */
        if ((fd = open(nom_tty, O_WRONLY)) < 0)
        {
            /* Ouverture impossible */
            message(1, "%s ne veut pas qu'on lui ecrive\n",
                    nom_ut);
            exit(0);
        }

        /* Envoi du message */
        for (argc -= 2, argv += 2; argc; argc--, argv++)
            message(fd, "%s ", *argv);
        message(fd, "\n  (%s)\n", getlogin());
        close(fd);

        /* Accuse d'emission */
        message(1, "message transmis a %s", nom_ut);
        message(1, " sur %s\n", nom_tty);
    }

    static void message(int fd, char *s1, char *s2)
    {
        char buf[128];

        sprintf(buf, s1, s2);
        write(fd, buf, strlen(buf));
    }
```

message.c

Les quelques lignes suivantes montrent diverses utilisations de cette commande.

```
$ message hibeme existe-t-il
hibeme n'existe pas
$
$ message casteran est-il connecte
casteran n'est pas connecte
$
$ message henry veut il un message
henry ne veut pas qu'on lui ecrive.
$ who | grep henry
henry    tty29   Aug  23 08:45
$ ls -l /dev/tty29
crwx-w----  1 henry      1,  29 Aug 23 08:45 /dev/tty29
$
$ message strandh un message pour tester
message transmis a strandh sur /dev/console
$
```

La première tentative porte sur un utilisateur qui n'est pas défini sur cette machine. Par conséquent, la fonction `getpwnam` échoue dans sa recherche et retourne 0. La seconde porte sur un utilisateur défini mais non connecté au moment ou la commande message s'exécute. Dans ce cas, c'est la recherche dans le fichier `/etc/utmp` qui échoue. La troisième tentative porte sur un utilisateur connecté, mais dont le terminal est protégé en écriture, par exemple au moyen de la commande `mesg -n`. Enfin, la dernière tentative s'adresse à un utilisateur connecté et dont le terminal est accessible en écriture. Sur son terminal, `/dev/console` s'affiche le message :

```
un message pour tester
  (achille)
$
```

Chapitre 12

Manipulation des processus Unix

12.1 Structure d'un processus

12.1.1 Contexte d'un processus

Un processus est le résultat du chargement d'un programme exécutable en machine. Le programme exécutable est un objet statique, tandis que le processus est un objet dynamique, pouvant évoluer tout au long de son existence. Il peut y avoir simultanément plusieurs occurrences d'un même exécutable en mémoire.

On appelle **contexte** d'un processus l'ensemble des informations qui le définissent: son numéro d'identification, sa priorité, l'adresse de la prochaine instruction à exécuter, les entrées-sorties en cours, etc. Une partie de ce contexte est cachée à l'utilisateur; il s'agit d'informations gérées par le système et stockées dans le noyau. La partie du contexte directement accessible à l'utilisateur est le **contexte utilisateur** du processus. Il est formé du code exécutable, des données et de la pile d'exécution. En langage d'assembleur, il est également possible d'accéder aux registres de la machine.

Bien qu'il ne soit pas présent dans le processus, le contexte système est partiellement accessible au moyen d'appels système. Certaines de ces informations sont fixées par le système et ne sont pas modifiables, comme par exemple le numéro d'identification du processus. D'autres, au contraire, peuvent être modifiées, soit directement au moyen d'appels système appropriés, soit par effet de bord de certains autres appels.

Nous allons maintenant passer rapidement en revue les principales informations du contexte système qui sont accessibles par programme.

Identification du processus

A sa création, tout processus reçoit un numéro unique qui est son identificateur. On le désigne généralement sous le terme de **pid**, contraction du terme anglo-saxon *process identifier*. Tout processus est créé par un autre processus, excepté le processus initial, de nom **swapper** et de *pid* 0, créé artificiellement au chargement du système.

Le *swapper* crée un processus appelé `init`, de *pid 1*, qui initialise le temps partagé. L'ensemble des processus existant à un moment donné forme un arbre dont la racine est le processus initial `swapper`.

Chaque processus connaît le numéro du processus qui l'a créé, appelé processus parent ou processus père. Cette information est désignée sous le terme de **ppid**, contraction du terme anglo-saxon *parent process identifier*.

Tout processus a accès à ses propres *pid* et *ppid*, au moyen des deux appels système **GETPID (2)** et **GETPPID (2)**. En voici un exemple très simple, utilisant également la commande **PS (1)**, qui fournit des informations sur les processus existants (par défaut, seulement les processus de l'utilisateur).

```
═══════════ pid.c ══════════════════════════════
#include <stdio.h>

main()
{
    printf("Je suis le processus %d ", getpid());
    printf("fils de %d\n", getppid());
    system("ps -l");
}
═════════════════════════════════ pid.c ═══════
```

Sur l'exemple d'exécution, nous avons simplifié le résultat de la commande `ps` pour faciliter la lecture de l'exemple.

```
$ pid
Je suis le processus 2214 fils de 1560
      F UID   PID  PPID       TT  TIME COMMAND
 408201 105  1552  1551       p0  0:26 -bash (bash)
   8001 105  1558  1552       p0 12:33 emacs -fn 9x15
 408201 105  1560  1559  ...  p1  0:07 -bash (bash)
   8001 105  2214  1560       p1  0:00 pid
   8001 105  2215  2214       p1  0:00 sh -c ps -l
      1 105  2216  2215       p1  0:00 ps -l
$
```

Le numéro d'un processus étant unique, il est classique de l'utiliser pour générer un identificateur, lorsqu'on veut éviter un conflit de nom. C'est par exemple le cas des noms de fichiers temporaires, qui doivent être tous différents. La technique appropriée consiste à former un nom dont le préfixe est propre à l'application et le suffixe est le numéro du processus.

```
═══════════ nomtemp.c ══════════════════════════
#include <stdio.h>
#include <stdlib.h>
#include "basename.h"

#define PREFIX "/tmp/"
#define LPID   5

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

```
	char *cmd = basename(argv[0]);
	char *nomtemp = malloc(strlen(PREFIX)
				+ strlen(cmd) + LPID + 1);

	sprintf(nomtemp, "%s%s%0*d", PREFIX, cmd, LPID, getpid());
	printf("\"%s\"\n", nomtemp);
}
```

nomtemp.c

On peut vérifier que deux occurrences différentes de la commande **nomtmp** utilisent des noms de fichier temporaire différents.

```
$  nomtemp ; nomtemp
"/tmp/nomtemp00259"
"/tmp/nomtemp00260"
$
```

L'ensemble des processus présents à tout moment en machine est partitionné en groupes. Par défaut, un nouveau processus appartient au même groupe que son père; ce groupe lui est attribué à sa création sous la forme d'un numéro appelé *pgrp*. Contrairement au numéro de processus, le numéro de groupe est modifiable (si cela n'était pas le cas, tous les processus appartiendrait à un même et unique groupe, celui du processus initial).

La modification du numéro de groupe se fait au moyen de l'appel système **SETPGRP(2)**. Cet appel s'utilise différemment sous SYSTEM V et sous BSD. Dans le premier cas, il s'utilise sans paramètre : le numéro du processus devient le nouveau numéro de groupe (celui-ci est alors assuré d'appartenir à un nouveau groupe, son *pid* étant unique). Dans le second, il est possible de préciser un numéro de groupe, et le numéro du processus à modifier. Le groupe d'un processus est utilisé lors de l'envoi de signaux, mécanisme présenté en 12.2.

Utilisateur et groupe propriétaire

Chaque processus reçoit à sa création un numéro d'utilisateur propriétaire qui est le même que celui de son père. Un utilisateur normal ne peut le modifier (si c'était le cas, il n'y aurait plus de possibilité de contrôle d'accès).

Le système gère en fait deux numéros, un numéro dit **effectif** et l'autre dit **réel**. Le numéro réel, ou *uid*, identifie l'utilisateur qui a créé le processus; le numéro effectif, ou *euid*, qui peut être différent du numéro réel, est celui utilisé en priorité par le système pour effectuer les contrôles d'accès.

Le numéro effectif est différent du numéro réel quand, dans le mode du fichier exécutable, est positionné un bit spécial appelé le *set uid* bit. Ce bit est positionnable au moyen de l'appel système **CHMOD(2)** (il s'agit du bit 04000 du mode d'accès – voir 11.1.2) ou au moyen de la commande **CHMOD(1)** (option u+s). Dans ce cas, l'identificateur effectif du processus est l'identificateur du propriétaire du fichier exécutable.

L'identificateur utilisé en priorité pour contrôler les accès étant l'identificateur effectif, il est par conséquent possible à un utilisateur d'attribuer ses propres accès à une commande qu'il possède, et ce quelque soit l'utilisateur qui la lance. Tous les utilisateurs qui ont le droit de lancer cette commande peuvent, en la faisant exécuter, accéder à des données qui leur sont interdites

sinon. Par exemple, certaines commandes du système, comme les commandes **SU (1)** ou **PASSWD (1)**, appartiennent au super-utilisateur `root` et ont leur *set uid* bit positionné.

```
$ ls -l /bin/passwd /bin/su
-rwsr-xr-x  3 root        24576 May  3  1989 /bin/passwd
-rwsr-xr-x  1 root        16384 Apr 25  1989 /bin/su
$
```

De cette façon, elles s'exécutent toujours en mode privilégié, quelque soit l'utilisateur qui les lance.

Il existe un mécanisme similaire avec le groupe propriétaire d'un processus (*gid* groupe réel et *egid* groupe effectif); le bit de marquage du fichier exécutable s'appelle le *set gid* bit (bit `02000` du mode d'accès et option **g+s** de la commande `chmod`).

Il est possible de connaître ces identificateurs au moyen des appels :

$$\begin{cases} \text{GETUID}(2) \\ \text{GETEUID}(2) \\ \text{GETGID}(2) \\ \text{GETEGID}(2) \end{cases}$$

Par exemple, on peut restreindre l'utilisation d'une commande à une liste d'utilisateurs donnée avec un test de la forme :

```
if (!autorise(getuid()))
{
    fprintf(stderr, "Sorry.\n");
    exit(1);
}
```

où `autorise` est une fonction vérifiant que l'identificateur reçu en paramètre appartient à la liste des identificateurs autorisés. On utilise ici le numéro d'utilisateur réel.

Signalerie

Un signal est un message qu'un processus peut envoyer à un autre processus sous certaines conditions. Lorsqu'un processus reçoit un signal, son exécution est suspendue; s'il existe un traitement associé à ce signal, le processus opère alors un déroutement vers ce traitement.

Par défaut, il n'y a pas de traitement spécifique associé à un signal; la réception d'un signal provoque dans ce cas généralement la terminaison du processus. Cette terminaison est alors qualifiée de **terminaison anormale**, par opposition à la terminaison normale, provoquée par l'exécution de l'appel **_EXIT (2)**. Un programme peut attacher des traitements à des signaux, ou tout simplement ignorer leur réception. La **signalerie** d'un processus est la liste de ces traitements; chaque traitement étant défini par une fonction, la signalerie d'un processus est un vecteur de pointeurs de fonctions indicé par les numéros de signaux. Nous présentons la mise en œuvre de ces mécanismes en 12.2.

Entrées-sorties

L'environnement d'entrées-sorties d'un processus est composé de la liste de ses descripteurs de fichiers (voir 10.1.2), certains descripteurs pouvant faire référence à des entrées-sorties initialisées.

Masque de création

Il s'agit du masque `umask` de création de fichier déjà présenté en 11.2.1.

Répertoires

Le répertoire courant d'un processus est celui utilisé par le système pour évaluer un chemin exprimé en relatif, c'est-à-dire ne commençant pas par le caractère /. Un processus hérite à sa création du répertoire courant de son père. Par conséquent, un processus lancé depuis l'interprète de commandes s'exécute *dans* le répertoire courant de l'utilisateur.

Un processus peut se déplacer dans l'arborescence des fichiers au moyen de l'appel système **CHDIR(2)**. Par exemple,

```
chdir("..")
```

permet de remonter dans le répertoire père du répertoire courant; il devient le nouveau répertoire courant. De même,

```
chdir(getenv("HOME"))
```

place le processus dans le répertoire d'accueil spécifié dans l'environnement du processus. Nous avons présenté, page 398, une fonction utilisant cet appel pour parcourir récursivement une arborescence, afin de la détruire.

Un processus possède également un répertoire racine personnalisé qui par défaut est la racine du système de fichiers. Il peut être modifié au moyen de l'appel **CHROOT(2)**. Le processus est alors *enfermé* dans une sous-arborescence du système de fichiers. Cela permet de définir des utilisateurs restreints n'ayant accès qu'à une partie des ressources de la machine.

Mesures

<sys/time.h> <sys/types.h>

Il s'agit de mesures de temps d'exécution, exprimées en 60$^{\text{ème}}$ de seconde. Ces mesures sont

- `tms_utime` : temps CPU consacré à l'exécution en mode utilisateur;
- `tms_stime` : temps CPU utilisé en mode système;
- `tms_cutime` : somme des temps utilisateurs du processus et de tous les fils qu'il a lancés et dont l'exécution est terminée;
- `tms_cstime` : somme des temps système du processus et de tous les fils qu'il a lancés et dont l'exécution est terminée.

Il est possible de connaître ces statistiques au moyen de la fonction `times`, implémentée sous la forme d'un appel système sous SYSTEM V et d'une fonction de bibliothèque sous BSD. Elle reçoit en paramètre l'adresse d'une structure

de type `struct tms`, définie dans le fichier `<sys/times.h>`, formée des quatre champs cités précédemment.

L'exemple suivant est celui du module `usertime` contenant deux fonctions, `usertime_reset` et `usertime_get`, permettant de mesurer un intervalle de temps utilisateur dans un processus en exécution. La première initialise une variable `utime`, locale au module, avec la valeur courante du temps utilisateur, et la seconde retourne la différence entre la valeur courante et la valeur sauvegardée dans `utime`, c'est-à-dire le temps écoulé depuis le dernier appel à `resetutime`.

```
========== usertime.c ==========
#include <sys/types.h>
#include <sys/times.h>

#include "usertime.h"

static int utime;
static struct tms tms;

void usertime_reset()
{
    times(&tms);
    utime = tms.tms_utime;
}

int usertime_get()
{
    times(&tms);
    return  tms.tms_utime - utime;
}
========== usertime.c ==========
```

On peut tester ces fonctions au moyen de la commande `test_usertime` suivante :

```
========== test_usertime.c ==========
#include <stdio.h>
#include <stdlib.h>

#include "usertime.h"

static void usage(char *);

main(int argc, char *argv[])
{
    int nb;
    int i;
    register int ri;
    int u;

    if (argc == 1)
        usage(argv[0]);

    nb = atoi(argv[1]);
    printf("Execution de %d iteration%s\n",
```

```
            nb, nb > 1 ? "s" : "");

        printf(" - sur une variable de pile: ");
        usertime_reset();
        for (i = nb; i > 0 ; i--)
            ;
        u = usertime_get();
        printf("%d/60 secondes (%4.2fs)\n", u, (float)u/60.0);

        printf(" - sur un registre: ");
        usertime_reset();
        for (ri = nb; ri > 0 ; ri--)
            ;
        u = usertime_get();
        printf("%d/60 secondes (%4.2fs)\n", u, (float)u/60.0);
    }

    static void usage(char *s)
    {
        fprintf(stderr, "Usage: %s entier\n", s);
        exit(1);
    }
```

test_usertime.c

L'exécution de cette commande fait afficher la différence de temps d'exécution entre *atoi(argv[1])* itérations effectuées sur une variable de pile et sur un registre. La session suivante contient deux exécutions, une compilée sans optimiseur de code et l'autre avec.

```
$ test_usertime 10000000
Execution de 10000000 iterations
 - sur une variable de pile: 243/60 secondes (4.05s)
 - sur un registre: 122/60 secondes (2.03s)
$
$ gcc -g -O test_usertime.c usertime.c -o test_usertime_O
$
$ test_usertime_O 10000000
Execution de 10000000 iterations
 - sur une variable de pile: 61/60 secondes (1.02s)
 - sur un registre: 60/60 secondes (1.00s)
$
```

12.1.2 Structure d'un programme exécutable

Un programme exécutable est un fichier contenant des instructions machine, et éventuellement des variables initialisées. Il est parfois terminé par une table de symboles, produite par l'éditeur de liens **LN(1)**, et donnant pour chaque symbole du programme l'adresse mémoire qu'il occupera durant l'exécution du processus. Chacune de ces parties est appelée un **segment** du programme.

Remarque 38 *Le code et les données d'un programme exécutable étant disjoints, tout processus* UNIX *est par défaut réentrant : plusieurs occurrences en*

mémoire d'un même exécutable partagent le même code.

Tout programme exécutable commence par un en-tête décrivant sa structure (taille des segments, présence ou non de la table des symboles...). Le format de cet en-tête est défini dans le fichier `<a.out.h>` sous la forme d'une structure appelée `struct exec` .

Exploitation de la table des symboles du noyau

La table des symboles d'un programme permet de connaître l'adresse d'implémentation d'une variable ou d'une fonction. Une application typique consiste à utiliser la table des symboles de l'exécutable du noyau UNIX pour lire une information directement dans la partie de la mémoire où il s'exécute. La mémoire d'exécution du noyau, ou *kernel memory*, est accessible au moyen du fichier spécial `/dev/kmem`, et l'exécutable du noyau s'appelle en général `/unix` ou `/vmunix`.

Le programme `av.c` suivant illustre ce fonctionnement; il accède à la mémoire d'exécution du noyau en ouvrant en lecture le fichier spécial `/dev/kmem` et se positionne au moyen de l'appel `lseek` à l'adresse du premier élément du vecteur `avenrun`. Celui-ci contient les statistiques de charge moyenne du système (*load average*) : nombre moyen de processus présents dans la file d'exécution depuis respectivement une, cinq et quinze minutes.

La version que nous donnons ici est celle tournant sur un Sun sous système Sun OS/4. Le vecteur `avenrun` est dans ce cas un vecteur de trois entiers longs, défini dans le fichier en-tête `<sys/kernel.h>`, et dont chaque élément code une statistique de charge multipliée par un coefficient. Ce coefficient est défini par une pseudo-constante, `FSCALE`, définie dans le fichier entête `<sys/param.h>`.

av.c

```
#include <sys/param.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/kernel.h>

#include <stdio.h>
#include <fcntl.h>

#include "concatener.h"

#define KMEM "/dev/kmem"

main(int argc, char *argv[])
{
    int fd = open(KMEM, O_RDONLY);

    if (fd == -1)
    {
        perror(concatener(argv[0], ":" KMEM, NULL));
        exit(1);
    }
    lseek(fd, (long) AVENRUN, 0);
    read(fd, avenrun, sizeof avenrun);
    printf(" Charge moyenne: %.2f-%.2f-%.2f\n",
```

```
            (float)avenrun[0] / (float)FSCALE,
            (float)avenrun[1] / (float)FSCALE,
            (float)avenrun[2] / (float)FSCALE);
}
```

av.c

La façon la plus simple de connaître l'adresse mémoire d'un symbole est d'utiliser la commande NM(1) listant la table des symboles d'un exécutable. Dans l'exemple qui suit, on récupère de cette façon la valeur du symbole

_avenrun[1]

avec laquelle on définit la pseudo-constante AVENRUN au moyen de l'option -D. Elle est utilisée dans le programme comme argument de l'appel lseek, pour effectuer le positionnement en lecture sur kmem.

```
$ nm /vmunix | grep _avenrun
0f081a38 B _avenrun
$ gcc -g av.c -DAVENRUN=0x0f081a38 -o av
$
```

L'accès à l'unité de mémoire kmem est normalement protégé en écriture et en lecture. L'exécution de la commande av produit alors l'erreur :

```
$ av
av:/dev/kmem: Permission denied
$
```

Il est bien sûr primordial d'empêcher aux utilisateurs d'accéder en écriture à /dev/kmem. Dans le cas contraire, il serait possible de modifier n'importe qu'elle ressource du système, comme la priorité ou le propriétaire d'un processus, voire tout simplement d'en altérer les données en provoquant une panne. L'accès en lecture doit également être protégé, car dans le cas contraire, il devient possible de lire toutes les ressources du noyau, et par conséquent d'*espionner* toute activité sur la machine sans contrôle des droits d'accès.

Certaines commandes, comme PS(1) ou W(1), ont cependant besoin d'accéder en lecture à /dev/kmem. Par exemple, la commande ps parcourt la table des processus du noyau pour en extraire des informations concernant les processus. La solution, dans ce cas, consiste comme on l'a déjà évoqué en 12.1.1 (page 411) à utiliser le mécanisme d'identificateur effectif. Une configuration généralement rencontrée est la suivante :

- un groupe fictif kmem est défini dans la liste des groupes d'utilisateurs;
- le fichier fichier spécial /dev/kmem est attribué au groupe kmem;
- chaque commande ayant à accéder à /dev/kmem, w, ps... est également attribuée au groupe kmem;
- le *setgid* bit est positionné sur ces commandes.

[1]Le caractère _ est ajouté devant chaque symbole par l'éditeur de liens. Cela permet d'éviter les conflits de nom entre les symboles du programme et des symboles prédéfinis de l'environnement (voir 12.1.4).

On peut vérifier cette configuration au moyen de la commande ls :

```
$ ls -lg /dev/kmem
crw-r----- 1 root  kmem    3,   1 Jun  7 16:23 /dev/kmem
$
$ ls -lg /bin/ps
-rwxr-sr-x 1 root  kmem     30528 Apr 25  1989 /bin/ps
$
```

L'installation de la commande av peut se faire de la manière suivante :

```
$ su
Password:
#
# ls -lg av
-rwxr-xr-x  1 achille  syntimag  24576 May 29 01:40 av
# chgrp kmem av
# chmod g+s av
# ls -lg av
-rwxr-sr-x  1 achille  kmem      24576 May 29 01:40 av
# C-d
$
```

Bien évidemment, seul le super-utilisateur est habilité a effectuer de telles manipulations, d'où l'utilisation de la commande **SU (1)**. Il est traditionnel de configurer son environnement de manière à changer son prompt en # lorsqu'on passe en mode super-utilisateur. Une fois cette manipulation effectuée, tout utilisateur a la possibilité d'exécuter la commande av : Pour vérifier le résultat de l'exécution de ce programme, on utilise la commande **W (1)** (BSD) qui affiche la charge courante de la machine.

```
$ av
 Charge moyenne: 0.63 0.45 0.48
$
$ w | grep load | grep -v grep
 5:23pm up 9 days, 4:11, 9 users, load average: 0.61,0.44,0.48
$
```

Les résultats sont légèrement différents car la charge du système évolue au cours du temps. En particulier, l'exécution des commandes av, w et grep influe sur les statistiques.

Recherche d'une information dans une table de symboles

`<nlist.h>`

La fonction **NLIST (3)** permet d'exploiter la table des symboles d'un exécutable depuis un programme. Elle reçoit en paramètres le nom du fichier à examiner et un vecteur de structures de type `struct nlist`. Cette structure, définie dans `<nlist.h>`, est composée, entre autres, d'un pointeur de caractères `n_name` et d'un entier sans signe `n_value`, contenant respectivement un nom de symbole et son adresse mémoire dans le processus.

Le vecteur transmis en paramètre à la fonction `nlist` est initialisé de la façon suivante :

- le champ `n_name` de chaque élément du vecteur pointe sur un nom de symbole à rechercher;
- le champ `n_name` du dernier élément est le pointeur nul.

Il ne faut pas oublier d'ajouter le caractère _ devant chaque symbole.

Pour chaque champ `n_name` contenant un symbole défini dans l'exécutable, la fonction `nlist` initialise les autres champs de la structure avec les valeurs correspondantes de la table des symboles. La commande `av` peut être récrite de la façon suivante :

```
av.2.c
#include <sys/param.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/kernel.h>

#include <stdio.h>
#include <fcntl.h>
#include <nlist.h>

#include "concatener.h"

#define UNIX "/vmunix"
#define KMEM "/dev/kmem"

main(int argc, char *argv[])
{
    int fd = open(KMEM, O_RDONLY);
    struct nlist n_list[] =
    {
        {"_avenrun", },
        {0, }
    };

    if (fd == -1)
    {
        perror(concatener(argv[0], ":" KMEM, NULL));
        exit(1);
    }
    if (nlist(UNIX, n_list) == -1)
    {
        perror(concatener(argv[0], ":" UNIX, NULL));
        exit(1);
    }
    lseek(fd, n_list[0].n_value, 0);
    read(fd, avenrun, sizeof avenrun);
    printf(" Charge moyenne: %.2f-%.2f-%.2f\n",
           (float)avenrun[0] / (float)FSCALE,
           (float)avenrun[1] / (float)FSCALE,
           (float)avenrun[2] / (float)FSCALE);
}
av.2.c
```

Suppression de la table des symboles

La présence de la table des symboles dans un exécutable est facultative. Pour diminuer la taille d'un exécutable, il est possible de la supprimer au moyen de la commande **STRIP(1)** :

```
$ ls -l av
-rwxr-xr-x  1 achille users       32768 Feb  4 14:51 av2
$ strip av ; ls -l av
-rwxr-xr-x  1 achille users       24576 Feb  4 15:21 av2
$ nm av
nm:  av:  no symbols
$
```

Il est également possible de demander à l'éditeur de liens de construire un exécutable sans table de symboles, avec l'option `-s`

12.1.3 Régions d'un processus

Un processus est rangé en mémoire dans plusieurs zones appelées **régions**, dont les trois principales sont

- la région contenant le code exécutable, ou **région du texte**;
- la région contenant les données ou **région des données**;
- la région contenant la pile d'exécution.

La figure 12.1 illustre le mécanisme de création d'un processus à partir d'un programme exécutable.

La région du texte est construite directement par recopie du segment texte de l'exécutable[2]. La construction de la région des données est un peu particulière. En effet, seules les données initialisées sont présentes dans le programme exécutable; la partie contenant les données non initialisées, encore appelée *BSS*[3], n'est construite qu'au lancement du processus, ceci pour ne pas occuper inutilement de la place dans l'exécutable.

Considérons les deux programmes suivants :

```
============ petit.c ============
main()
{
}
============ petit.c ============
```

```
============ gros.c ============
int vecteur[100000];
```

[2]Dans le cas où il existe déjà une occurrence de cette commande en train de s'exécuter, les régions texte des deux processus sont partagées. Il n'y a pas, dans ce cas, de recopie du segment texte, mais simplement incrémentation d'un compteur de références.

[3]Ce nom provient d'un code opération, utilisé sur des machines IBM de la série 70, signifiant *Bloc Started by Symbol*, et permettant d'allouer une zone représentée par sa taille dans le fichier objet.

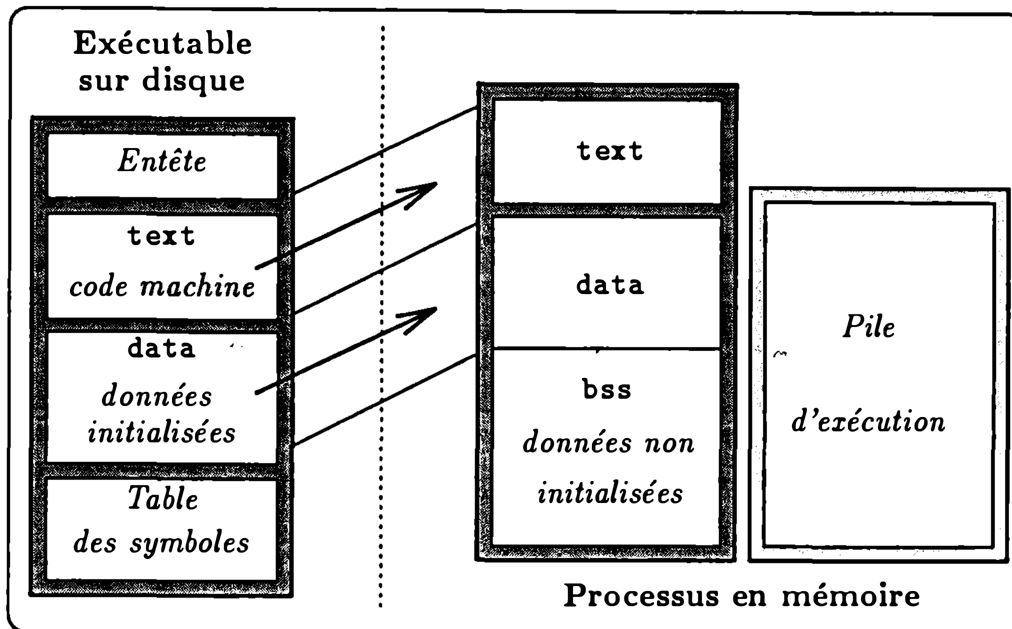


Figure 12.1: Création d'un processus.

```
main()
{
}
```

gros.c

On peut vérifier que les exécutables de ces deux programmes ont la même taille :

```
$ ls -l gros petit
-rwxr-xr-x  1 achille     24576 Aug 26 04:03 gros
-rwxr-xr-x  1 achille     24576 Aug 26 04:06 petit
$
```

La taille de la région des données peut être modifiée au cours de l'exécution du processus. Cela s'effectue au moyen des deux appels système :

- **BRK(2)** modifiant la taille de la région des données avec un paramètre exprimé en absolu,
- **SBRK(2)** la modifiant relativement à la taille courante.

C'est avec ces appels que sont écrites les fonctions d'allocation dynamique de mémoire présentées en 7.3. On trouvera des exemples d'utilisation de ces appels page 425 et page 429.

L'en-tête d'un fichier exécutable décrit son format; on y trouve, entre autres, les informations suivantes :

- **a_text** : taille du segment de texte;
- **a_data** : taille du segment des données initialisées;
- **a_bss** : taille des données non initialisées;
- **a_syms** : taille de la table des symboles.

Plusieurs pseudo-fonctions sont définies dans le fichier `a.out.h`; elles calculent l'adresse de début et la taille des segments en prenant en compte le format du fichier exécutable. Ce format est codé par un mot placé en tête du fichier et appelé le *nombre magique*.

12.1.4 Démarrage et terminaison d'un processus

L'exécution d'un processus débute sur une adresse appelée le *point d'entrée*. Cette adresse est contenue dans le champ `a_entry` de l'en-tête de l'exécutable. Par défaut, le point d'entrée est la première instruction d'une fonction prédéfinie appelée `start`, dont le rôle est :

- de récupérer les arguments et l'environnement du processus; cette opération dépend des machines et peut éventuellement être écrite en assembleur (par exemple, quand il est nécessaire d'accéder directement aux registres du processeur);
- d'initialiser la variable globale `_environ` qui pointe sur le vecteur contenant la liste des variables d'environnement;
- d'appeler la fonction `main` avec les paramètres `argc`, `argv` et `envp`;
- de provoquer la terminaison du processus lors du retour éventuel de la fonction `main`.

L'arrêt d'un processus est provoqué par l'exécution de l'appel système _EXIT(2). On peut aussi utiliser la fonction EXIT(3) qui vide les tampons d'entrées-sorties de haut niveau avant d'appeler `_exit`. Le rôle de cet appel est de libérer les ressources occupées par le processus, et de transmettre certaines informations de terminaison à son père.

C'est la seule façon correcte pour un processus de terminer son exécution. On parle dans ce cas de **terminaison normale**. La terminaison peut également être causée par l'envoi d'un signal (voir 12.2) au processus. Si le processus ne gère pas explicitement la réception de ce signal, il est tué et les ressources qu'il occupe sont aussitôt libérées.

Il s'agit alors d'une terminaison anormale, parfois accompagnée de la production d'un fichier de nom `core` contenant l'image mémoire du processus au moment de la terminaison. Ce fichier peut alors être exploité au moyen d'un débogueur. La fonction `start` est définie dans le fichier `crt0.o`; elle est insérée, par l'éditeur de liens, dans tout programme exécutable. L'éditeur de liens ne rajoute pas le caractère _ devant le nom du symbole `start`; de cette façon, il ne risque pas d'y avoir de conflit avec un symbole défini par le programmeur.

Voici un exemple de table des symboles d'un fichier `crt0.o` :

```
$ nm /lib/crt0.o
         U __exit
00000278 D _environ
         U _exit
         U _main
00000000 T start
$
```

On y trouve

- la définition de la variable globale `environ` (`D` pour segment `DATA`);

- la définition de la fonction `start` (`T` pour segment `TEXT`);
- une référence à la fonction `main`, indéfinie dans ce fichier objet (`U` pour `UNDEFINED`); elle provient de l'appel à `main` effectué dans la fonction `start`;
- des références aux fonctions `exit` et `_exit` de la bibliothèque standard et de l'interface avec les appels système.

Voici un exemple de fonction `start`. Au moment de la création du processus, la pile d'exécution contient le nombre d'arguments et l'adresse de début de la liste des arguments et variables d'environnement. Le premier paramètre de la fonction correspond au premier argument.

```
char **environ;

start(char *arg)
{
    int argc = (int)(&arg)[-1];

    environ = &arg + argc + 1;
    exit(main(argc, &arg, environ));
}
```

La valeur transmise en paramètre à `exit` est la valeur de retour du processus. Cette valeur est tronquée et seuls en sont conservés les huit bits de poids faible. Par défaut, un programme termine avec la valeur 0. Il est possible de modifier cette valeur en appelant explicitement la fonction `exit`, ou en effectuant un `return` dans la fonction `main`.

L'interprète de commandes récupère cette valeur et la traite comme un booléen. La valeur par défaut 0 associée à *vrai*, et toute autre valeur à *faux*. La commande `exit_with` de l'exemple qui suit illustre ce mécanisme.

```
════════ exit_with.c ════════
#include <stdlib.h>

main(int argc, char *argv[])
{
    if (argc > 1)
        exit(atoi(argv[1]));
}
════════ exit_with.c ════════
```

Nous utilisons ci-dessous la structure de contrôle de test du shell, dont la syntaxe est :

```
if ⟨commande⟩
then ⟨commande⟩
else ⟨commande⟩
fi
```

```
$ if exit_with 0
> then echo vrai
> else echo faux
> fi
vrai
```

```
$
$ if exit_with 1
> then echo vrai
> else echo faux
> fi
faux
$
```

L'interprète place dans une variable réservée de nom ? la valeur de retour de la dernière commande exécutée, convertie en une chaîne de caractères :

```
$ exit_with 33
$ echo $?
33
$ echo $?
0
$
```

12.1.5 Sauvegarde d'image mémoire

Il est possible de recréer un programme exécutable à partir d'un processus en exécution. Cette opération particulièrement intéressante est connue sous le nom de *sauvegarde d'image mémoire.* Elle permet de sauvegarder l'environnement complet d'un processus à un moment donné de son exécution. Lorsque le nouvel exécutable ainsi créé est relancé, son exécution reprend avec l'environnement sauvegardé.

Nous allons, dans cette section, présenter une version simple de sauvegarde d'image mémoire, dont le principe est le suivant :

- le nouvel exécutable est construit sans table de symbole (version minimale);
- le code machine du nouvel exécutable est celui du processus; il suffit donc de connaître l'adresse de début et la taille de la région du texte en mémoire et de la recopier telle quelle dans le fichier exécutable à créer;
- la région des données initialisées du nouvel exécutable est formée de l'ensemble de toutes les données du processus; si on connaît l'adresse de début et la taille de la région des données en mémoire, comme dans le cas précédent il suffit de la recopier dans le fichier;
- le nouvel en-tête est formé à partir de l'ancien, en apportant les modifications suivantes :
  * la taille du segment BSS (champ **a_bss**), c'est-à-dire des données non initialisées, est ramenée à zéro;
  * la nouvelle taille du segment DATA (champ **a_data**), c'est-à-dire des données initialisées est celle de la région des données toute entière : par conséquent, dans le nouvel exécutable, toutes les données sont des données initialisées;
  * la taille de la table des symboles (champ **a_syms**) est ramenée à zéro.

Remarque 39 *Il existe d'autres formes de sauvegarde d'image mémoire, comme par exemple celle, très performante, mise en œuvre dans la programmation de l'éditeur GNU Emacs. On pourra, à ce sujet, consulter le fichier* `unexec.c` *contenu dans le répertoire* `src` *de la distribution de ce logiciel.*

Nous allons donner un exemple de mise en œuvre du mécanisme énoncé plus haut, s'appuyant sur un format de fichier exécutable commun à UNIX BSD et à un certain nombre d'implémentations SYSTEM V, et dans lequel l'exécutable est composé des quatre segments représentés sur la figure 12.1. Il existe d'autres formats d'exécutables, comme par exemple le format COFF (*common object file format*) sous certaines implémentations SYSTEM V, mais ils dérivent en général de ce format de base.

Nous allons tout d'abord définir la fonction `creaout`, recevant en paramètres l'en-tête du processus courant et un nom de fichier dans lequel elle effectue la sauvegarde d'image mémoire. Elle initialise l'en-tête du nouvel exécutable et le recopie dans le fichier avec les régions *text* et *data* du processus.

L'implantation mémoire des régions est donnée par les trois pseudo-fonctions DEBUT_TEXT, DEBUT_DATA et TAILLE_TEXT définies dans le fichier en-tête `svaout.h`. L'appel `sbrk`, invoqué avec la valeur 0 en paramètre, permet de connaître l'adresse de fin de la région des données du processus. Les pseudo-constantes DEBUT_DATA, DEBUT_TEXT, et TAILLE_TEXT sont également définies dans le fichier `svaout.h`. Ces définitions dépendent de la machine sur laquelle s'exécute ce programme; les renseignements utiles pour l'écrire se trouvent dans la page de manuel A.OUT(4) sous SYSTEM V et A.OUT(5) sous BSD. A titre d'exemple, nous donnons celui utilisé sur un Sun 3/60 sous système *OS/3* :

```
/* Debut du texte en memoire */
#define DEBUT_TEXT(x) (N_TXTADDR(x) + (sizeof(x)-N_TXTOFF(x)))

/* Taille du texte en memoire*/
#define TAILLE_TEXT(x) ((x).a_text - (sizeof(x)-N_TXTOFF(x)))

/* Debut des donnees en memoire */
#define DEBUT_DATA(x) (N_DATADDR(x))

/* Debut du segment texte dans l'executable */
#define DEBUT_SEG_TEXT(x) (sizeof(x))
```

Les pseudo-fonctions N_TXTADDR, N_TXTOFF, et N_DATADDR sont définies dans le fichier `a.out.h`. La fonction remplaçant le fichier exécutable du processus par sa sauvegarde est facile à écrire. Nous en donnons ici une version avec la fonction `svaout`, appelable avec le chemin de l'exécutable du processus courant. Elle en lit l'en-tête, détruit le fichier exécutable et appelle la fonction `creaout` pour en reconstruire une nouvelle version. Il est nécessaire de détruire explicitement le fichier au moyen de `unlink` car il n'est pas possible d'ouvrir en écriture un fichier exécutable pour lequel il existe un processus en cours d'exécution.

```
svaout.c
#include <a.out.h>
#include <fcntl.h>
```

```
#include "svaout.h"

/* Debut du texte en memoire */
#define DEBUT_TEXT(x) (N_TXTADDR(x) + (sizeof(x)-N_TXTOFF(x)))

/* Taille du texte en memoire*/
#define TAILLE_TEXT(x)  ((x).a_text - (sizeof(x)-N_TXTOFF(x)))

/* Debut des donnees en memoire */
#define DEBUT_DATA(x)   (N_DATADDR(x))

/* Debut du segment texte dans l'executable */
#define DEBUT_SEG_TEXT(x) (sizeof(x))

/*
  Creation du fichier 'nomaout' avec les segments du processus
  et l'en-tete 'entete'
 */
creaout(struct exec entete, char *nomaout)
{
    char *sbrk(int);
    int fd = open(nomaout, O_WRONLY|O_CREAT|O_TRUNC, 0755);

    if (fd < 0)
        return 0;

    /* Fabrication de la nouvelle entete */
    entete.a_bss = 0;
    entete.a_syms = 0;
    entete.a_data = sbrk(0) - (char *) DEBUT_DATA(entete);
    write(fd, &entete, sizeof entete);

    /* Recopie du texte */
    lseek(fd, DEBUT_SEG_TEXT(entete), 0);
    write(fd, DEBUT_TEXT(entete), TAILLE_TEXT(entete));

    /* Recopie des donnees */
    write(fd, DEBUT_DATA(entete), entete.a_data);
    close(fd);
    return 1;
}

/*
  Sauvegarde de l'image memoire du processus ('nomaout' est
  le nom de l'executable)
 */
svaout(char *nomaout)
{
    struct exec entete;
    int fd = open(nomaout, O_RDONLY);

    /* Lecture de l'entete de l'executable */
    if (fd < 0)
        return 0;
    read(fd, &entete, sizeof entete);
    close(fd);
    unlink(nomaout);
    return creaout(entete, nomaout);
```

```
}
```
svaout.c

Voici un exemple d'utilisation de ces fonctions à travers le programme tsv.c. Il se compose d'une boucle itérant les actions suivantes :

- lecture d'une chaîne de caractères et recopie dans une zone mémoire allouée dynamiquement,
- chaînage dans une liste au moyen d'une cellule, elle aussi allouée dynamiquement.

Si la chaîne lue est un caractère ? le programme imprime la liste de toutes les chaînes de la liste. La lecture d'une chaîne vide provoque la terminaison du programme avec sauvegarde d'image mémoire. On peut alors le relancer, et on retrouve toutes les données lues lors des précédentes exécutions. La lecture s'effectue au moyen de la fonction lire_chaine présentée page 222.

tsv.c

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

#include "lire_chaine.h"

static void lister();
static void ajouter(char *);

struct liste
{
    char         *chaine;
    struct liste *suivant;
};
struct liste premier = { 0, 0 };
struct liste *dernier = &premier;

main(int argc, char *argv[])
{
    for (;;)
    {
        char *chaine;

        chaine = lire_chaine("-> ");
        if (*chaine == '\0')
        {
            svaout(argv[0]);
            exit(0);
        }
        if (chaine[0] == '?')
            lister();
        else
            ajouter(chaine);
    }
}

/* Impression de la liste */
static void lister()
{
    struct liste *l;
```

```
    if (premier.suivant == 0)
        printf(" (NIL)\n");
    else
    {
        for (l=premier.suivant; l; l = l->suivant)
            printf(" \"%s\"\n", l->chaine);
    }
}

/* Ajout de la chaine s dans la liste */
static void ajouter(char *s)
{
    struct liste *l = (struct liste *) malloc(sizeof *l);

    l->chaine = s;
    l->suivant = 0;
    dernier->suivant = l;
    dernier = l;
}
```
tsv.c

L'extrait de session suivant montre trois appels successifs à `svaout`. La première fois, la liste est vide car le programme vient juste d'être compilé. Lors de la seconde exécution, la liste contient toutes les chaînes saisies précédemment. La troisième exécution s'effectue à partir de la sauvegarde de la seconde et bénéficie des deux précédentes saisies.

```
$ gcc -g tsv.c svaout.o lire_chaine.o -o tsv
$ tsv
-> ?
 (NIL)
-> Un
-> Deux
->
$
$ tsv
-> ?
 "Un"
 "Deux"
-> Trois
->
$
$ tsv
-> Quatre
-> Cinq
-> ?
 "Un"
 "Deux"
 "Trois"
 "Quatre "
 "Cinq"
->
$
```

12.1.6 Chargement dynamique

Une autre opération très intéressante, et en quelque sorte symétrique de la sauvegarde d'image mémoire, est le chargement dynamique du code exécutable d'une fonction dans un processus en cours d'exécution.

Cette opération est facilitée par une option de l'éditeur de liens, l'option `-A` permettant une édition de liens incrémentale. L'éditeur de liens utilise dans ce cas la table des symboles d'un exécutable déjà créé (à priori celui qui effectue le chargement dynamique). Le fichier résultant a la structure d'un fichier exécutable mais ne contient que le code et les données des nouvelles fonctions. L'option `-T` permet de spécifier à partir de quelle adresse de base doit être effectuée la translation d'adresses; ce sera l'adresse à laquelle la fonction sera chargée en mémoire. L'option `-N` génère du code impur dans lequel les données sont situées directement à la suite du code. De cette manière, on minimise la taille du programme à charger. Enfin, l'option `-e` permet de redéfinir le point d'entrée qui est par défaut la fonction `_start` (voir 12.1.4). Dans ce cas, le point d'entrée est le nom de la fonction à charger.

Si l'on fait les hypothèses suivantes :

- le fichier *fich*`.o` contient le code objet de la fonction *fnct*;
- l'éditeur doit utiliser la table des symboles du fichier *tsym*;
- le fichier exécutable *fich* doit être chargé à partie de l'adresse mémoire *orig*;

la construction du fichier *fich* s'effectue ainsi :

`ld -T` *orig* `-A` *tsym* `-N -e` *_fnct* `-o` *fich fich*`.o`

Ce travail est effectuer par la fonction `reloger` du programme `charger.c`

L'adresse `orig` est la première adresse libre du programme, c'est-à-dire l'adresse donnée par l'expression

`sbrk(0)`

Lorsque l'exécutable chargeable est construit, il suffit de lire son en-tête pour calculer la taille du programme à charger : c'est la somme de la taille du code et de la taille des données. La région des données est alors augmentée de la taille du programme à charger :

`sbrk(header.a_text + header.a_data + header.a_bss)`

Il ne reste plus qu'à lire les segments de texte et de données de l'exécutable à partir de l'adresse `orig`. L'adresse de la fonction en mémoire est la valeur du point d'entrée de l'exécutable chargé; c'est le champ `a_entry` de l'en-tête. La fonction `charger` effectue ce traitement.

```
════════ charger.c ════════════════════════
#include <stdio.h>
#include <fcntl.h>
#include <a.out.h>

char *sbrk(int);
```

```
static void reloger(char *, char *, char *, void *);

/*
 Chargement dynamique de la  fonction  'fonction'  contenue
 dans le fichier 'fonction.o'; 'charger' retourne l'adresse
 en memoire de la fonction chargee.
 */
int (* charger(char *executable, char *fonction))()
{
    char *origine = sbrk(0);
    int fd;
    int taille;
    struct exec exec;

    reloger(executable, fonction, fonction, origine);

    /* Calcul de la taille du programme a charger */
    if ((fd = open(fonction, O_RDONLY)) == -1
        || read(fd, &exec, sizeof exec) < sizeof exec)
    {
        fprintf(stderr,
                "Impossible de charger %s\n", fonction);
        return 0;
    }
    taille = N_SYMOFF(exec) - N_TXTOFF(exec);

    /* Extension de la region des donnees */
    if (sbrk(exec.a_text+exec.a_data+exec.a_bss)==(char *)-1)
    {
        fprintf(stderr,
                "Programme %s trop gros\n", fonction);
        return 0;
    }

    /* Chargement de l'executable */
    lseek(fd, (long) N_TXTOFF(exec), 0);
    read(fd, origine, taille);
    close(fd);
    return (int (*) ()) exec.a_entry;
}

/*
   Parametres
     - fichier contenant la table des symbole
     - fichier a construire
     - symbole du point d'entree
     - adresse de chargement dans le processus
 */
static void reloger(char *symboles, char *fichier,
                    char *fonction, void *origine)
{
    char commande[256];

    sprintf(commande,
            "ld -Bstatic -A %s  -T %x -N -e _%s  -o %s %s.o",
            symboles,
            (long) origine,
            fonction,
```

```
            fichier,
            fichier);
    system(commande);
}
=================================== charger.c ========
```

Dans l'exemple qui suit, la commande `tchr` effectue le chargement d'une fonction dont le nom lui est passé comme premier argument. Cette fonction est supposée être appelable avec deux paramètres entiers et retourner une valeur entière. Elle est appelée avec les second et troisième arguments de la commande `tchr`.

```
======== tchr.c ====================================
#include <stdio.h>

#include "charger.h"

static void usage();

main(int argc, char **argv)
{
    int(*operation)();

    if (argc < 4)
        usage(argv[0]);

    operation = charger(argv[0], argv[1]);
    printf("%s de %s et de %s = %d\n",
           argv[1], argv[2], argv[3],
           (* operation) (atoi(argv[2]), atoi(argv[3])));
}

static void usage(char *argv0)
{
    fprintf(stderr, "Usage: %s operation n1 n2\n", argv0);
    exit(1);
}
==================================== tchr.c ========
```

Nous allons tester la commande `tchr` avec le chargement de deux fonctions, `addition` et `multiplication`, respectivement contenues dans les fichiers objet `addition.o` et `multiplication.o`.

```
======== addition.c ================================
int addition(int a, int b)
{
    return a + b;
}
================================= addition.c ========
```

```
======== multiplication.c ==========================
int multiplication(int a, int b)
{
```

```
    return a*b;
}
```
multiplication.c

Il est maintenant possible de compiler de nouvelles fonctions et de les charger directement dans le module de calcul.

```
$ gcc -g -c addition.c
$ gcc -g -c multiplication.c
$
$ charger addition 10 19
addition de 10 et de 19 = 29
$
$ charger multiplication 10 19
multiplication de 10 et de 19 = 190
$
```

Il est également possible de charger une fonction utilisant des objets définis dans le programme de chargement, grâce au mécanisme d'édition de liens incrémentale. En voici un exemple avec la fonction `division` utilisant la variable globale `argv0` pour imprimer un message d'erreur :

division.c

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

extern char *argv0;

int division (int a, int b)
{
    if (b == 0)
    {
        fprintf(stderr, "%s: division par zero\n", argv0);
        return HUGE_VAL;
    }
    return a/b;
}
```
division.c

Le symbole `argv0` est bien sûr indéfini dans le fichier objet `division.o`. Sa valeur est attribuée lors de l'édition de lien dynamique à partir de la table des symboles de `tchr`.

```
$ gcc -g -c division.c
$ nm division.o
         U __iob
         U _argv0
00000018 T _division
         U _exit
         U _fprintf
00000000 t gcc_compiled.
$
$ tchr division 35 6
```

```
division de 35 et de 6 = 5
$ tchr division 35 0
tchr: division par zero
$
```

12.2 Gestion des signaux

12.2.1 Signalerie Unix

Un signal est un message prioritaire destiné à un processus. Il peut être envoyé par le noyau UNIX, ou par un autre processus. Les signaux émis par le noyau le sont à la suite d'une anomalie survenue pendant l'exécution du processus (utilisation d'une adresse mémoire illégale, décrochement de modem...)

Chaque signal est codé par un entier positif. Il existe selon les versions d'UNIX entre une vingtaine et une trentaine de signaux, chacun correspondant à un événement particulier. Par défaut, l'envoi d'un signal à un processus provoque une terminaison anormale, avec pour certains signaux, sauvegarde de l'image mémoire du processus dans un fichier appelé `core`. La création de ce fichier est accompagnée de l'émission du message `core dumped` sur la sortie d'erreur standard.

Il est possible d'ignorer la réception d'un signal ou de l'associer à un traitement spécifique. Afin d'éviter qu'un processus ne puisse être rendu *indestructible* par masquage de tous les signaux, certains d'entre eux, et notamment le signal `KILL` de numéro 9, ne sont ni masquables ni redéfinissables. Excepté s'il s'exécute en mode super-utilisateur, un processus ne peut envoyer de signal qu'à des processus ayant le même numéro d'utilisateur que lui.

Les principales causes d'émission d'un signal sont les suivantes :

- frappe d'un caractère d'interruption depuis la console attachée au processus; ce sont les caractères *interrupt* et *quit* (en général `C-c` ou `DEL` et `C-\`), et sous BSD le caractère *stop* (en général `C-z`); lorsqu'il reçoit un de ces caractères, le pilote *tty* le répercute aussitôt au processus qui lui est attaché sous la forme d'un signal;
- exécution de la commande **KILL(1)** depuis l'interprète de commandes; la commande

  `kill -`⟨*n*⟩ ⟨*pid*⟩

  envoie le signal numéro ⟨*n*⟩ au processus numéro ⟨*pid*⟩; le signal `KILL` ayant le numéro 9, `kill -9` ⟨*pid*⟩ garantit la terminaison du processus ⟨*pid*⟩;
- envoi d'un signal par un processus utilisateur au moyen de l'appel système **KILL(2)**;
- événement provoquant une interruption matérielle : exécution d'une instruction réservée, erreur d'arithmétique, accès à une adresse mémoire illégale...
- événements système divers : écriture sur un tube fermé en lecture, mécanisme d'alarme...

Le tableau 12.1 donne la liste des principaux signaux. Ils sont définis dans le fichier en-tête `<signal.h>`.

Nom		Numéro		Signification
		BSD	Sy.V	
SIGHUP		1	1	Décrochement de ligne (*hangup*)
SIGINT		2	2	Caractère *interrupt* au clavier
SIGQUIT	⋆	3	3	Caractère *quit* au clavier
SIGILL	⋆	4	4	Instruction illégale
SIGTRAP	⋆	5	5	Traçage d'un processus
SIGABRT	⋆	6	6	Fonction standard `abort`
SIGEMT	⋆	7	7	Instruction EMT
SIGFPE	⋆	8	8	Erreur arithmétique
SIGKILL	o	9	9	Signal de terminaison (non masquable)
SIGBUS	⋆	10	10	Violation d'une protection (*bus error*)
SIGSEGV	⋆	11	11	Accès mémoire illégal (*Segmentation violation*)
SIGSYS	⋆	12	12	Argument d'appel système incorrect
SIGPIPE		13	13	Ecriture sur un tube fermé en lecture
SIGALRM		14	14	Envoi d'une alarme par l'horloge
SIGTERM		15	15	Signal de terminaison par défaut
SIGTSTP		18	–	Caractère *stop* au clavier
SIGCONT		19	–	Reprise après un *stop*
SIGCHLD		20	18	Terminaison d'un processus fils
SIGUSR1		30	16	Signal utilisateur numéro 1
SIGUSR2		31	17	Signal utilisateur numéro 2

⋆ : Production d'un fichier `core`
o : Ne peut être ni récupéré ni masqué

Tableau 12.1: Principaux signaux UNIX

12.2.2 Envoi de signal

L'envoi d'un signal est effectué au moyen de l'appel système **KILL(2)**. L'appel `kill` de SYSTEM V contient toutes les formes d'envoi de signal UNIX; sous BSD, il est également nécessaire d'utiliser un second appel, **KILLPG(2)**, l'appel `kill` étant moins complet.

Un processus s'exécutant en mode normal peut envoyer un signal à

1) un processus ayant même identificateur utilisateur effectif;
2) tous les processus ayant même identificateur utilisateur effectif;
3) tous les processus ayant pour identificateur de groupe l'identificateur du processus;
4) tous les processus d'un groupe de processus donné et ayant même identificateur utilisateur.

Le tableau 12.2 résume les différents comportements de l'appel `kill`.

Le programme `autokill` suivant s'envoie à lui-même le signal dont il reçoit le numéro en argument.

	`kill(`p`, `s`)`: envoi du signal s sous SYSTEM V	sous BSD
$p > 0$	au processus de *pid* p	au processus de *pid* p
$p = 0$	aux processus de *pgrp* π	aux processus de *pgrp* π
$p = -1$	si $u = 0^{(*)}$ à tous les processus non système sinon aux processsus d'*uid* u	si $u = 0^{(*)}$ à tous les processus non système sinon —
$p < -1$	à tous les processus de *pgrp* $-p$	`killpg (`$-p$`, `*sig*`)`
Identification du processus émetteur: *pid*$=\pi$, *euid*$=u$, *pgrp*$=g$		

(*) super-utilisateur

Tableau 12.2: Description de l'appel `kill(`*pid*`,`*sig*`)`

```
══════════ autokill.c ══════════════════════════════════════
#include <stdio.h>

void usage(char *);

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

    if (argc != 2)
        usage(argv[0]);

    sig = atoi(argv[1]);
    printf("%d: ", sig);
    fflush(stdout);
    kill(getpid(), sig);
}

void usage(char *s)
{
    fprintf(stderr, "Usage: %s signal\n", s);
    exit(1);
}
══════════════════════════════════════ autokill.c ══════════
```

Dans l'exemple qui suit, le programme **autokill** est successivement invoqué avec les valeurs de 1 à 15. On vérifie ainsi le comportement par défaut d'un processus sur réception d'un signal. Le message d'erreur affiché est produit par l'interprète de commandes lorsqu'il est informé que son fils **autokill** effectue une terminaison anormale sur réception d'un signal. Voici tout d'abord une exécution sous l'interprète shell **bash** :

```
$ for S in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
> do
>     autokill $S
> done
1: Hangup
```

```
2:
3: Quit - core dumped
4: Illegal instruction - core dumped
5: Trace/BPT trap - core dumped
6: abort - core dumped
7: EMT trap - core dumped
8: Floating exception - core dumped
9: Killed
10: Bus error - core dumped
11: Memory fault - core dumped
12: Bad system call - core dumped
13:
14: Alarm call
15: Terminated
$
```

Les messages d'erreur peuvent varier suivant les interprètes. Voici par exemple la même session effectuée sous l'interprète *cshell* `tcsh` :

```
% foreach S (1 3 4 5 6 7 8 9 10 11 12 13 14 15)
? autokill $S
? end
1: Hangup
3: Quit (core dumped)
4: Illegal instruction (core dumped)
5: Trace/BPT trap (core dumped)
6: IOT trap (core dumped)
7: EMT trap (core dumped)
8: Floating exception (core dumped)
9: Killed
10: Bus error (core dumped)
11: Segmentation fault (core dumped)
12: Bad system call (core dumped)
13: Broken pipe
14: Alarm clock
15: Terminated
%
```

Le signal 2 n'apparaît pas dans la liste car il provoque, en *cshell*, la terminaison de la boucle.

Il est possible d'invoquer `kill` avec le numéro de signal 0. Dans ce cas, l'appel vérifie seulement que le processus récepteur existe, et qu'il est possible de lui envoyer un signal. Cette option permet de tester l'existence et l'appartenance d'un processus.

kill0.c

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

void usage(char *);

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

```
{
    int pid;

    if (argc != 2)
        usage(argv[0]);
    pid = atoi(argv[1]);

    if (kill(pid, 0) == -1)
    {
        switch (errno)
        {
          case ESRCH:
            printf("Processus %d inexistant.\n", pid);
            break;
          case EPERM:
            printf("Processus %d inaccessible.\n", pid);
            break;
        }
    }
    else
        printf("Processus %d accessible.\n", pid);
}

void usage(char *s)
{
    fprintf(stderr, "Usage: %s pid\n", s);
    exit(1);
}
```

kill0.c

Le programme est invoqué successivement avec un numéro de processus dont l'*uid* est différent (le processus `init` numéro 1), un numéro de processus de même *uid* (le shell courant) et un numéro de processus inexistant :

```
$ kill0 1
Processus 1 inaccessible.
$ kill0 $$
Processus 2994 accessible.
$ kill0 6666
Processus 6666 inexistant.
$
```

12.2.3 Récupération de signal

`<signal.h>`

Un signal peut être ignoré ou associé à une fonction qui sera exécutée lors de sa réception. Cela peut se faire au moyen de la fonction `signal`, implémentée comme appel système sous SYSTEM V (**SIGNAL(2)**) et comme fonction de la bibliothèque sous BSD (**SIGNAL(3)**). Cette fonction a également été intégrée à la bibliothèque ANSI, avec un jeu de signaux plus restreints. La fonction `signal` s'utilise avec deux paramètres qui sont le numéro d'un signal et le traitement à associer à ce signal; la spécification de traitement peut être :

– un pointeur de fonction : cette fonction est appelée sur réception du signal;
– la constante **SIG_IGN** : le signal est ignoré;
– la constante **SIG_DFL** : le traitement par défaut est restauré.

Lorsqu'une fonction est associée à un signal, le traitement du signal s'effectue de la façon suivante :

1) interruption de l'exécution du processus;
2) appel de la fonction de traitement du signal;
3) reprise de l'exécution au point d'interruption.

La fonction de traitement de signal est appelée avec le numéro du signal reçu en paramètre. Voici un exemple très simple dans lequel un processus déroute toute réception de signal sur une fonction de nom **handler**, puis répète indéfiniment un appel à **PAUSE(2)**; cet appel endort le processus jusqu'à réception d'un signal.

```
═══════ handler.c ═══════════════════════
#include <signal.h>
void handler(int);

#include <stdio.h>

void handler(int);

main()
{
    int i;

    for (i = 1; i < NSIG; i++)
        signal(i, handler);

    for(;;)
        pause();
}

void handler(int s)
{
    printf("--> recu signal %d\n", s);
}
═══════════════════════ handler.c ═══════
```

Le nombre de signaux, **NSIG**, est une pseudo-constante définie dans le fichier en-tête **<signal.h>**. On peut tester ce programme en le lançant en détaché, et en lui envoyant des signaux au moyen de la commande **kill**.

```
$ handler &
[1] 3230
$
$ kill -2 3230
--> recu signal 2
$ kill -15 3230
--> recu signal 15
$ kill -77 3230
kill: bad signal spec '77'
$
```

```
$ kill -9 3230
[1]+  Killed                 handler
$
```

La fonction de traitement de signal ne peut recevoir de paramètre spécifique, ni retourner de valeur. Elle peut par contre modifier des variables globales et influer ainsi sur la suite de l'exécution du processus.

L'exemple suivant est un programme qui boucle en affichant chaque seconde la valeur d'une variable globale, la variable `compteur`. Elle récupère les signaux `SIGINT` et `SIGQUIT` associés aux caractères d'interruption clavier *intr* et *quit* (en général `C-c` et `C-\`).

signal.c

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

void interruption();
void arret();

int pas_fini = 1;
int compteur = 0;

main()
{
    signal(SIGQUIT, arret);
    signal(SIGINT,  interruption);
    while (pas_fini)
    {
        printf("%d\n", compteur);
        sleep(1);
    }
}

/* Traitement du signal SIGQUIT */
void arret()
{
    printf("Reception de SIGQUIT: arret\n");
    pas_fini = 0;
}


/* Traitement du signal SIGINT */
void interruption()
{
    printf("Reception de SIGINT: incrementation\n");
    compteur++;
}
```

signal.c

La réception de `SIGQUIT` provoque le positionnement d'une variable globale, la variable `pas_fini`, et la terminaison du programme. La réception de `SIGINT` provoque l'incrémentation de la variable `compteur`.

```
$ signal
```

```
0
0
^CReception de SIGINT: incrementation
1
^CReception de SIGINT: incrementation
2
2
2
^\Reception de SIGQUIT: arret
$
```

La récupération d'un signal peut être utilisée conjointement au mécanisme de sauvegarde de contexte d'exécution (voir 7.9). Le programme précédent peut être récrit de la façon suivante :

sig2.c

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

void interruption();
void arret();

int compteur = 0;
jmp_buf avant_boucle;

main()
{
    signal(SIGQUIT, arret);
    signal(SIGINT,  interruption);
    if (setjmp(avant_boucle) == 0)
    {
        for (;;)
        {
            printf("%d\n", compteur);
            sleep(1);
        }
    }
}

/* Traitement du signal SIGQUIT */
void arret()
{
    printf("Reception de SIGQUIT: arret\n");
    longjmp(avant_boucle, 1);
}


/* Traitement du signal SIGINT */
void interruption()
{
    printf("Reception de SIGINT: incrementation\n");
    compteur++;
}
```

sig2.c

Les deux programmes `sig1` et `sig2` ont un comportement équivalent. On préfèrera cependant la seconde forme qui est plus adaptée au traitement d'exceptions.

Remarque 40 *Sous les dernières versions de* BSD, *ce mécanisme a été enrichi (voir la page* **SIGVEC(2)** *du manuel* UNIX BSD*); cependant des programmes utilisant ces nouvelles fonctionnalités ne sont pas compatibles avec la version actuelle de* SYSTEM V.

12.2.4 Gestion d'une alarme

`<signal.h>`

Le système UNIX gère un mécanisme d'alarme, fonctionnant sur le principe d'un compte à rebours. La mise en œuvre d'une alarme est la suivante :

- à un temps t, un processus P active une alarme avec un délai Δt;
- au temps $t + \Delta t$, le processus P reçoit le signal `SIGALARM`.

Le processus a, entre temps, la possibilité de désarmer l'alarme.

Le positionnement d'une alarme est géré par l'appel système **ALARM(2)** qui reçoit en paramètre le délai, exprimé en secondes, au bout duquel le signal doit être envoyé. La fonction `alarm_a` de l'exemple suivant positionne une alarme à déclencher à une date exprimée en heures, minutes et secondes. Elle convertit le temps courant et le temps d'activation de l'alarme en secondes; la différence entre ces deux valeurs donne le délai à fournir à l'appel `alarm`.

Le programme testant cette fonction utilise l'appel **PAUSE(2)** déjà évoqué page 438, et la fonction `hms` présentée page 238.

alarm_a.c

```
#include <signal.h>

#include "hms.h"

void alarme();
void alarm_a(int, int, int);

main(int argc, char *argv[])
{
    int h, m, s;

    signal(SIGALRM, alarme);
    alarm_a(atoi(argv[1]), atoi(argv[2]), atoi(argv[3]));
    hms(&h, &m, &s);
    printf("Je m'endors a %d:%d:%d\n", h, m, s);
    pause();
    hms(&h, &m, &s);
    printf("Je me reveille a %d:%d:%d\n", h, m, s);
}

void alarm_a(int h_alrm, int m_alrm, int s_alrm)
{
    int h, m, s;
    int temps_en_secondes;
    int delai;
```

```
    hms(&h, &m, &s);
    temps_en_secondes = 3600*h + 60*m + s;
    delai = 3600*h_alrm + 60*m_alrm + s_alrm - temps_en_secondes;
    if (delai < 0)
        delai += 3600*24;
    alarm(delai);
}

void alarme()
{
    printf("Reception du signal SIGALRM\n");
}
```
alarm_a.c

On peut contrôler le bon déroulement de ce programme en encadrant son exécution par deux appels à la commande **DATE(1)**.

```
$ date
Tue Jun  1 00:10:48 MET DST 1993
$
$ alarm_a 00 11 10 ; echo ; date
Je m'endors a 0:10:59
Reception du signal SIGALRM
Je me reveille a 0:11:10

Tue Jun  1 00:11:11 MET DST 1993
$
```

Une telle fonction peut être utilisée pour réveiller un processus, ou pour interrompre son déroulement à un temps donné. Une autre utilisation possible du mécanisme d'alarme est la sauvegarde de résultats intermédiaires au cours d'un très long calcul. De cette façon si le travail est stoppé, par exemple à cause d'un arrêt du système, il est possible de reprendre les calculs avec les valeurs de la dernière sauvegarde. La sauvegarde peut être effectuée au moyen de la fonction de sauvegarde d'image mémoire présentée en 12.1.5.

Le programme suivant, simplifié à l'extrême, illustre la mise en œuvre de ce principe. La fonction `main` initialise un calcul, et met en place une alarme, récupérée par la fonction `onalrm`. Elle lance ensuite un calcul, sur l'exemple une simple boucle. Toutes les `DELAI` secondes (3 secondes sur l'exemple), la réception du signal `SIGALRM` provoque une sauvegarde des résultats intermédiaires (ici, l'affichage à l'écran de la variable `i`).

alarm_sv.c

```
#include <signal.h>
void onalrm();

#define DELAI 3

unsigned long i;
void initialise();
void calcule();
void sauve();
```

```
main()
{
    signal(SIGALRM, onalrm);
    initialise();
    alarm(DELAI);
    calcule();
    printf("Calcul termine\n");
}

/* Initialisation des calculs */
void initialise()
{
    i = 0;
}

/* Le gros calcul... */
void calcule()
{
    while (i += 128)
        ;
}

/* Reception du signal, et reactivation de l'alarme */
void onalrm()
{
    sauve();
    alarm(DELAI);
}

/* Sauvegarde des calculs intermediaires */
void sauve()
{
    printf("Sauvegarde de i : %u\n", i);
}
```

alarm_sv.c

Le programme est lancé au moyen de la commande **TIME(1)** qui imprime les statistiques de temps d'exécution d'une commande. L'exécution de alarm_sv dure 15.6 secondes. Il y a 5 sauvegardes de résultats intermédiaires, au rythme d'une toutes les 3 secondes.

```
$ time alarm_sv
Sauvegarde de i : 848897024
Sauvegarde de i : 1698339456
Sauvegarde de i : 2549312256
Sauvegarde de i : 3396233216
Sauvegarde de i : 4247144064
Calcul termine
       15.6 real        15.2 user         0.0 sys
$
```

12.3 Création de processus

Un processus P peut créer un nouveau processus de deux façons différentes :

- par **duplication** : le processus P crée une copie de lui-même;
- par **substitution** : le processus P charge en mémoire un nouvel exécutable qui prend sa place.

Par conséquent, le lancement par un processus P d'un nouveau processus Q à partir de l'exécutable E, s'effectue en deux temps :

1) duplication du processus P : la copie de P est un nouveau processus P';
2) libération du contexte utilisateur P', régions de texte, données et pile, et chargement à sa place des segments de l'exécutable E; du point de vue du système, ce nouveau processus Q est identique à P'; en particulier, il possède le même *pid*.

Ces deux opérations sont réalisées par deux appels systèmes, la duplication par l'appel **FORK(2)**, et le remplacement par l'appel système **EXECVE(2)**. Ce dernier appel peut être mis en œuvre sous différentes formes, grâce aux fonctions répertoriées dans la page **EXECL(3)** du manuel sous BSD et **EXEC(2)** sous SYSTEM V. Dans ce qui suit, nous désignerons sous le terme générique `exec` l'une quelconque de ces formes.

12.3.1 Substitution d'un processus

Lorsqu'un processus exécute l'appel `exec`, il charge un programme exécutable en mémoire en conservant le même environnement système. Du point de vue du système, il s'agit toujours du même processus. En particulier, il occupe la même entrée dans la table des processus et conserve le même numéro d'identification. Du point de vue de l'utilisateur, l'opération est beaucoup plus radicale : un processus qui exécute l'appel `exec` disparaît; à sa place un nouveau processus, disposant du même environnement système, débute son exécution.

L'interface de l'appel système `exec` est composée des six fonctions

```
execl
execv
execle
execve
execlp
execvp
```

Elles diffèrent par le nombre et le format de leurs paramètres. Les paramètres nécessaires au lancement d'un nouvel exécutable sont :

- le chemin de cet exécutable dans le système de fichiers;
- la liste des arguments à lui transmettre : `argv[0]...`;
- la liste de ses variables d'environnement : `envp[0]...`

Sur les six fonctions d'appel, quatre d'entre elles s'emploient sans le paramètre d'environnement; elles transmettent au nouveau processus une copie des variables d'environnement du processus courant. Les deux fonctions de base sont `execl` et `execv`, dont les paramètres ont la forme suivante :

```
execl(char *⟨chemin⟩, char *⟨argv0⟩, ..., char *⟨argvn⟩, 0)
execv(char * ⟨chemin⟩, char **⟨argv⟩)
```

Dans le premier cas, les arguments de la commande à lancer sont fournis sous la forme d'une liste terminée par un pointeur nul; dans le second, ils le sont sous la forme d'un vecteur de pointeurs de caractères dont chaque élément pointe sur un argument, le vecteur étant terminé par un pointeur nul. Ce second format est identique à celui du vecteur `argv` reçu par la fonction `main` de tout programme. Il n'est pas nécessaire que le nom du fichier exécutable et celui du processus résultant de son lancement soient les mêmes.

Voici deux façons différentes de faire exécuter la commande **PS (1)** avec en argument l'option `-l`; la première utilise la fonction `execl` :

```
═══════ execl.c ═══════════════════════
#include <stdio.h>
#include <stddef.h>

main()
{
    printf("[%d]: chargement de ps par execl\n",
           getpid ());
    execl("/bin/ps", "PS", "-l", NULL);
    printf("Jamais par ici\n");
}
═══════════════════════ execl.c ═══════
```

et la seconde la fonction `execv` :

```
═══════ execv.c ═══════════════════════
#include <stdio.h>

char *argvec[] = { "PS", "-l", 0 };

main()
{
    printf("[%d]: chargement de ps par execv\n", getpid ());
    execv("/bin/ps", argvec);
    printf("Jamais par ici\n");
}
═══════════════════════ execv.c ═══════
```

Dans chaque cas, le nom donné au processus est `PS`, comme on peut le vérifier lors de l'exécution du programme. On constate également qu'à chaque fois, l'identificateur du processus lancé est celui du processus lanceur.

```
$ execl
[21141]: chargement de ps par execl
      F UID   PID  PPID   ...   STAT TT  TIME COMMAND
 400201 105  3630  3628   ...   S    p0  0:08 -bash
      1 105 21141  3630   ...   R    p0  0:00 PS -l
$
$ execv
[21144]: chargement de ps par execv
```

```
       F UID   PID  PPID ... STAT TT  TIME COMMAND
  400201 105  3630  3628 ... S    p0  0:08 -bash
       1 105 21144  3630 ... R    p0  0:00 PS -1
$
```

Le processus qui exécute la substitution, **21141** dans le premier cas et **21144** dans le second, remplace son code et ses données par ceux de la commande ps; une nouvelle pile d'exécution est initialisée, et l'exécution se poursuit sur la première instruction de la commande ps.

La forme **execv** de l'appel **exec** est très utile pour lancer une commande reçue en argument. Considérons l'exemple de la commande **lancer** qui, si elle est invoquée sous la forme

$$\texttt{lancer}\ \langle m_1\rangle\ \langle m_2\rangle\ldots\ \langle m_k\rangle$$

fait exécuter la commande $\langle m_1\rangle$ avec les arguments

$$\langle m_2\rangle\ldots\langle m_k\rangle$$

Le vecteur **argv** fourni au programme **lancer** vaut :

$$\begin{pmatrix} \texttt{"lancer"} \\ \langle m_1\rangle \\ \langle m_2\rangle \\ \vdots \\ \langle m_k\rangle \end{pmatrix}$$

La commande **lancer** effectue le lancement de la commande par

$$\texttt{execv(argv[1], argv+1)}$$

qui équivaut à

$$\texttt{execv(}\langle m_1\rangle\texttt{, \{}\langle m_1\rangle\texttt{, }\langle m_2\rangle\texttt{, ..., }\langle m_k\rangle\texttt{\})}$$

En voici une version qui affiche sur sa sortie standard la commande à lancer, puis effectue le lancement de la façon que nous venons de voir.

```
lancer.c
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <math.h>
#include "basename.h"
#include "formater.h"

static void entete_message(char *);

main(int argc, char *argv[])
{
    char *argv0 = basename(argv[0]);

    if (argc == 1)
    {
        entete_message(argv0);
        printf("rien a lancer\n");
```

```
        exit(0);
    }
    entete_message(argv0);
    printf("chargement de \"%s\"", argv[1]);
    if (argc > 2)
    {
        int c = argc-2;
        char **v = argv+2;

        printf(" avec ");
        while (c--)
        {
            printf("%s", *v);
            v++;
            if (c)
                printf(", ");
        }
    }
    printf("\n");
    execv(argv[1], argv+1);
}

static void entete_message(char *argv0)
{
    static char *prefix = NULL;

    if (prefix == NULL)
    {
        prefix = formater(Str, argv0, Str, "(",
                          Int, getpid(), Str, "): ", Eop);
    }
    printf("%s", prefix);
}
```

lancer.c

La première ligne de la session suivante, `echo *.c`, permet de visualiser la liste des noms de fichiers construite par le shell avec le motif `*.c`.

```
$ echo *.c
execl.c execv.c
$ lancer /bin/ls -i *.c
lancer(439): chargement de "/bin/ls" avec -i, execl.c, execv.c
74835 execl.c   74897 execv.c
$
$ lancer lancer lancer lancer
lancer(440): chargement de "lancer" avec lancer, lancer
lancer(440): chargement de "lancer" avec lancer
lancer(440): chargement de "lancer"
lancer(440): rien a lancer
$
```

Sous l'interprète de commandes, il existe un mécanisme de recherche automatique de l'exécutable d'une commande parmi une liste de répertoires spécifiques. Cette liste est propre à chaque utilisateur; elle est contenue dans la

variable d'environnement `PATH`. Ainsi, bien que la commande `ls` soit située dans le répertoire `/bin`, il n'est pas nécessaire de taper explicitement `/bin/ls` pour la faire exécuter. Il suffit de taper `ls`, le répertoire `bin` faisant partie par défaut de la liste des répertoires d'exécutables.

Il est possible de bénéficier de ce mécanisme en utilisant la commande `sh` avec une option particulière, l'option `-c`. Lorsqu'il est invoqué sous la forme

$$\texttt{sh -c } \langle par_1 \rangle \ldots \ \langle par_k \rangle$$

le shell traite la liste $\langle par_1 \rangle \ldots \ \langle par_1 \rangle$ comme s'il s'agissait d'une commande lue sur son entrée standard, et rend la main aussitôt l'exécution de cette commande terminée.

Voici un exemple simple dans lequel une commande est lue sous la forme d'une chaîne de caractères sur l'entrée standard, puis exécutée en utilisant ce procédé.

```
=========== sh_c.c ===========================
#include <stdio.h>
#include <stdlib.h>

main()
{
    char commande[1024];

    printf("-> ");
    gets(commande);
    execl("/bin/sh", "(sh)", "-c", commande, 0);
}
=========================================== sh_c.c ======
```

Cet exemple d'exécution utilise la commande `lancer` définie page 447.

```
$ sh_c
-> ps
  PID TT STAT  TIME COMMAND
  122 p4 I     1:11 -bash
 4592 p5 S     0:00 (sh) -c ps
 4593 p5 R     0:01 ps
$
$ sh_c
-> ls *.c
lancer.c    execl.c     execv.c     sh_c.c
$
$ sh_c
-> lancer sh_c
lancer(4589): chargement de "sh_c"
-> lancer sh_c
lancer(4590): chargement de "sh_c"
-> inconnu
(sh): inconnu: not found
$
```

Il est également possible d'utiliser les fonctions `execlp` et `execvp` qui invoquent l'interprète `sh`.

Par défaut, le processus lancé reçoit une copie de l'environnement du processus qui opère la substitution. Les fonctions `execle` et `execve` permettent de construire un nouvel environnement au processus substitué. Elles acceptent comme dernier argument la liste des variables d'environnement, sous une forme identique à celle du vecteur `envp` fourni à toute fonction `main` comme troisième argument (voir 7.5).

Voici un exemple utilisant la fonction `execle` pour lancer un shell en lui créant l'environnement:

$$\begin{cases} \texttt{PATH} : \text{"} : \texttt{/bin} : \texttt{/usr/bin} : \text{"} \\ \texttt{CC} \quad : \text{"}\texttt{gcc}\text{"} \\ \texttt{PS1} \quad : \text{"}[\langle ps1 \rangle]\text{"} \end{cases}$$

où $\langle ps1 \rangle$ est le prompt du shell appelant.

```
═════════ execle.c ══════════════════════════════════
#include <stdlib.h>
#include <stdio.h>
#include <stddef.h>

#include "concatener.h"

char *envpnt[] =
{
    "PATH=:/bin:/usr/bin:",
    "CC=gcc",
    NULL,
    NULL
};

main()
{
    envpnt[2] = concatener("PS1=[", getenv("PS1"),
                           "] ", NULL);
    execle("/bin/sh", "(un shell)", NULL, envpnt);
}
══════════════════════════════════════ execle.c ═════
```

Le nouveau prompt est construit à partir du prompt exporté de l'environnement shell, contenu dans la variable `PS1`. D'autres variables sont redéfinies par le shell lui-même.

```
$ execle
[$ ] set
CC=gcc
IFS=
PATH=:/bin:/usr/bin:
PS1=[$ ]
PS2=>
[$ ] execle
[[$ ] ] ps -l
 UID   PID  PPID  TT   TIME COMMAND
1000  5006  5005          p1  0:02 /usr/new/bash
1000  5026  5006          p1  0:00 sh
1000  5039  5026   ...    p1  0:00 (un shell)
```

```
1000  5040  5039         p1  0:00 (un shell)
1000  5041  5040         p1  0:00 ps -l
[[$ ] ]
C-d
[$ ] ps -l
 UID   PID  PPID        TT  TIME COMMAND
1000  5006  5005         p1  0:02 /usr/new/bash
1000  5026  5006    ...  p1  0:00 sh
1000  5039  5026         p1  0:00 (un shell)
1000  5042  5039         p1  0:00 ps -l
[$ ]
C-d
$
```

Le processus `execle` de numéro `5039` s'est remplacé par un shell en lui définissant un nouvel environnement utilisateur.

12.3.2 Duplication d'un processus

Contrairement à la substitution, le second mécanisme de création de processus, la **duplication** d'un processus existant provoque la création d'un nouveau processus. Ce nouveau processus possède son propre numéro d'identification, et l'ancien processus continue d'exister. La duplication est assurée par l'appel système **FORK(2)**. Lorsqu'un processus P_i exécute l'appel `fork`, le système crée une copie de P_i, différant seulement par le numéro d'identification. Cette copie effectuée, il existe en machine un nouveau processus P_j, partageant le même texte que P_i, et dont les données, la pile d'exécution et l'environnement système sont identiques à ceux de P_i. Le processus P_j commence son exécution au moment du retour de l'appel `fork` : *un seul processus effectue l'appel, et deux processus effectuent le retour.*

La fonction `fork` retourne une valeur différente dans le processus père P_i et dans le fils P_j. Cette valeur est zéro dans le fils, et le *pid* du fils dans le père. L'utilisation typique de l'appel `fork` est la suivante :

```
if ((pid = fork()) == 0)
{
    /* Instruction du fils */
    ...
}
else if (pid > 0)
{
    /* Instruction du pere */
    ...
}
else
    /* Erreur */
```

Le système ne peut créer qu'un nombre fini de processus. Cette limite, correspond à la taille de la table des processus dans le noyau du système. De même, le nombre de processus qu'un utilisateur peut faire exécuter simultanément est limité. L'appel `fork` échoue si l'une ou l'autre de ces deux bornes est atteinte. Il peut également échouer s'il n'y a plus assez de mémoire (centrale

+ *swap*) pour créer le nouveau processus.

Lorsque l'exécution de l'appel `fork` est terminée, les deux processus s'exécutent en parallèle. Considérons le programme suivant dans lequel un processus se duplique, puis le père et le fils affichent des messages.

```
fork.c
#include <math.h>
#include <stdlib.h>

#include "formater.h"

static void message(char *);
static char *argv0;

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

    argv0 = argv[0];
    message("debut du programme");
    pid = fork();
    switch (pid)
    {
      case -1:
        perror(argv[0]);
        exit(1);
        /* NOBREAK */

      case 0:
        message("/f/ retour de fork avec 0");
        break;

      default:
        {
            char *s = formater(Str, "/p/ retour de fork avec ",
                               Int, pid, Eop);
            message(s);
            free(s);
            break;
        }
    }
    message("fin du programme");
}

void message(char *s)
{
    printf("%s(=%d): %s\n", argv0, getpid(), s);
}
fork.c
```

Comme on peut le voir dans l'exemple qui suit, il est possible que le retour du `fork` se produise d'abord dans le processus fils, le processus père étant alors en attente d'exécution.

```
$ fork
fork(4783): debut du programme
```

```
fork(4784): /f/ retour du fork avec 0
fork(4783): /p/ retour du fork avec 4784
fork(4784): fin du programme
fork(4783): fin du programme
$
$ fork
fork(4785): debut du programme
fork(4785): /p/ retour du fork avec 4786
fork(4785): fin du programme
fork(4786): /f/ retour du fork avec 0
fork(4786): fin du programme
$
```

Le programme précédent est exécuté deux fois de suite. La première fois, le processus père porte le numéro 4783 et le fils 4784. Le message préfixé par /p/ (resp. /f/) est émis par le processus père (resp. fils). Le premier message est affiché une seule fois, l'instruction correspondante se situant avant la duplication. Par contre le dernier message est affiché deux fois, une fois par chaque processus.

L'exécution de ce programme est *non-déterministe*; comme on peut le constater, la seconde exécution produit un résultat différent de la première. Pour rendre ce programme déterministe, il est nécessaire de synchroniser les deux processus; il est par exemple possible de forcer le père à attendre la terminaison de son fils, au moyen de l'appel système **WAIT(2)** que nous détaillerons dans la section suivante. Un processus exécutant l'appel wait est endormi jusqu'à la terminaison de l'un de ses fils.

Il est très courant que le processus fils exécute un exec aussitôt créé. C'est en effet la façon standard de lancer un nouveau processus :

1) duplication d'un processus existant;
2) remplacement du fils par un nouveau processus.

Un exemple classique et très facile à mettre en œuvre est l'empilement d'un shell depuis un programme.

```
═══════════ shell.c ═══════════════════════════════════
#include <stdio.h>

static void shell(void);

main()
{
    shell();
    printf("Fin du test\n");
}

static void shell()
{
    if (fork())
        wait(0);
    else
    {
        chdir(getenv ("HOME"));
        execl("/bin/sh", "-sh", 0);
```

```
    }
}
```

═══════════════ shell.c ═══════

Ici nous utilisons la particularité du shell d'exécuter les commandes contenues dans le fichier d'initialisation

`~/.profile`

lorsque l'argument `argv[0]` commence par le caractère tiret.

```
$ cat ~/.profile
echo execution du fichier .profile
PS1="sh: "
$
$ shell
execution du fichier .profile
sh:
sh: ps -l
      F UID   PID  PPID ... STAT TT  TIME COMMAND
 400201 105  3630  3628 ... S    p0  0:11 -bash
      1 105 19376  3630 ... S    p0  0:00 shell
      1 105 19378 19376 ... S    p0  0:00 -sh
      1 105 19438 19378 ... R    p0  0:00 ps -l
sh:
sh: C-d
Fin du test
$  ps -l
      F UID   PID  PPID ... STAT TT  TIME COMMAND
 400201 105  3630  3628 ... S    p0  0:11 -bash
      1 105 19456  3630 ... R    p0  0:00 ps -l
$
```

Les processus existant aux différentes étapes de cette exécution sont les suivants :

1) avant l'exécution de `fork` :
   – le processus numéro 19376 de nom `shell` – actif –
2) avant l'exécution de `execl` par le fils :
   – le processus numéro 19376 de nom `shell` – endormi –
   – le processus numéro 19378 également de nom `shell` – actif –
3) après l'exécution de `execl` et avant `C-d` :
   – le processus numéro 19376 de nom `shell` – endormi –
   – le processus numéro 19378 de nom `-sh` – actif –
4) après `ps -l` :
   – le processus numéro 19376 de nom `shell` – endormi –
   – le processus numéro 19378 de nom `-sh` – endormi –
   – le processus numéro 19438 de nom `ps` – actif –
5) après `C-d` :
   – le processus numéro 19376 de nom `shell` – actif –

Une autre illustration de ce mécanisme est la fonction **SYSTEM(3)** présentée en 7.8 et dont voici une version simplifiée :

```
=========== systeme.c ===========
#include <stddef.h>

int systeme(char *s)
{
    int pid = fork();

    switch(pid)
    {
      case -1:
        return -1;
        /* NOBREAK */

      case 0:
        execl("/bin/sh", "(sh)", "-c", s, NULL);
        return 0;
        /* NOBREAK */

      default:
        return wait(0);
        /* NOBREAK */
    }
}
=========== systeme.c ===========
```

12.3.3 Héritage du contexte système

Lors de la duplication d'un processus, le processus fils hérite de tout le contexte système de son père (signalerie, entrées-sorties, identificateurs de propriétaire et de groupe, répertoire courant...).

Lors d'un remplacement, la signalerie est réinitialisée, les adresses des fonctions attachées aux signaux ne correspondant plus à rien dans le nouveau processus. Par contre, le nouveau processus hérite du reste du contexte système de son père, et en particulier de l'environnement d'entrées-sorties.

C'est ce principe d'héritage qui rend naturel le mécanisme de redirection. Considérons par exemple le programme suivant :

```
=========== tracer.c ===========
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>

#define FICHIER_TRACE "TRACE"

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

    close(1);
    open(FICHIER_TRACE, O_WRONLY|O_CREAT|O_APPEND, 0644);
```

```
    printf("\n$");
    for (i = 1; i < argc; i++)
        printf(" %s", argv[i]);
    printf("\n");
    fflush(stdout);

    execvp(argv[1], argv+1);
}
```
tracer.c

Il lance une commande reçue en argument après avoir redirigé sa sortie standard dans un fichier de nom TRACE et affiché un message. La redirection s'effectue en fermant l'entrée-sortie associée au descripteur numéro 1, puis en exécutant une ouverture en écriture, avec les options O_CREAT et O_APPEND. L'appel open attribue le premier descripteur libre à cette entrée-sortie, c'est-à-dire le descripteur 1 qui vient d'être libéré par l'instruction précédente.

La troisième instruction du programme effectue une écriture au moyen de la fonction `printf`, donc sur la sortie standard. La sortie standard étant redirigée, c'est dans le fichier TRACE que s'effectue l'écriture. La commande passée en argument est lancée par execvp. Elle s'exécutera dans ce nouveau contexte système, c'est-à-dire avec sa sortie standard redirigée dans le fichier TRACE. Voici quelques exemples d'exécution de ce programme.

```
$
$ rm -f TRACE
$ tracer date
$ tracer ps
$ tracer ls -l TRACE
$
$ cat TRACE

$ date
Mon Jul 19 23:32:53 MET DST 1993

$ ps
  PID TT STAT  TIME COMMAND
 9205 p1 IW    0:03 /usr/new/bash -login
 9217 p1 IW    0:05 twm
 9232 p1 IW    0:07 emacs
 9235 p1 IW    0:00 /usr/local/emacs/etc/wakeup 60
 9236 p2 S     0:02 -bash (bash)
 9265 p2 R     0:00 ps

$ ls -l TRACE
-rw-r--r--  1 achille       564 Jul 19 23:44 TRACE
$
```

C'est le même principe d'héritage des entrées-sorties qui permet à deux processus parents de communiquer au moyen d'un tube. Le tube est initialisé avant l'appel à `fork`. Lors de la duplication, le processus fils héritant des entrées-

sorties de son père, les deux processus ont la possibilité de lire et d'écrire dans le même tube.

Les quelques instructions suivantes montrent l'initialisation d'un tube utilisé par le processus père pour écrire au processus fils.

```
pipe(fd);
if (fork())
{
    close(fd[0]);
    pere(fd[1]);
}
else
{
    close(fd[1]);
    fils(fd[0]);
}
```

Dans chaque processus, on ferme le descripteur inutilisé. Cela assure un comportement correct du mécanisme de détection de fin de fichier (tube vide et plus de processus producteur). On peut tester ce programme en définissant deux fonctions `pere` et `fils`, comme c'est le cas dans le fichier `tube.c`.

```
=========> tube.c ===========================================
#include <stdio.h>

main()
{
    int fd[2];
    void pere();
    void fils();

    pipe(fd);
    if (fork())
    {
        close(fd[0]);
        pere(fd[1]);
    }
    else
    {
        close(fd[1]);
        fils(fd[0]);
    }
}

void pere(int fd)
{
    char buffer[256];

    printf("[pere-%d]: envoi de mon pid\n", getpid());
    sprintf(buffer, "%d", getpid());
    write(fd, buffer, strlen(buffer));
}

void fils(int fd)
{
    printf("[fils-%d]: recu \"", getpid());
```

```
    for (;;)
    {
        char c;

        if (read(fd, &c, 1) < 1)
            break;
        putchar(c);
    }
    printf("\" dans le tube\n");
}
```

tube.c

Chaque fonction reçoit en paramètre le descripteur qu'elle utilise pour effectuer son entrée-sortie. La fonction père écrit la chaîne de caractères correspondant à son *pid* dans le tube et termine son exécution. Le processus fils lit dans le tube caractère par caractère jusqu'à détection de fin de fichier, en recopiant sur sa sortie standard chaque caractère lu. L'exécution du programme donne un résultat de la forme :

```
$ tube
[pere-4700]: envoi de mon pid
[fils-4701]: recu "4700" dans le tube
$
```

Nous reviendrons plus en détail sur la mise en œuvre des mécanismes de communication par tubes en 13.2.

Chapitre 13

Communications interprocessus

13.1 Synchronisation de processus par signaux

13.1.1 Attente d'une terminaison

Nous avons vu en 12.1.4 que lorsqu'un processus effectue une terminaison normale, sa valeur de retour est transmise au système par l'appel `_exit`. Une terminaison anormale est provoquée par la réception d'un signal qui n'est ni masqué ni récupéré. Dans ce cas, la valeur de terminaison du processus est indéfinie, et la terminaison est associée à un code d'erreur composé du numéro du signal ayant provoqué cette terminaison, et d'un indicateur signalant si un fichier `core` a été produit.

La valeur et le code d'erreur d'un processus qui se termine peuvent être obtenus par le processus qui l'a créé au moyen de l'appel système **WAIT(2)** :

```
int wait(int *⟨terminaison⟩)
```

Un processus exécutant cet appel est endormi jusqu'à la terminaison d'un de ses fils. Lorsque cela se produit il est réveillé, et l'appel `wait` retourne le *pid* du processus fils en question. Le paramètre ⟨*terminaison*⟩ est passé par référence; il reçoit la valeur de retour et le code d'erreur du processus.

Si le processus ne possède pas de fils vivant lors de l'exécution de l'appel `wait`, ce dernier retourne immédiatement la valeur −1. Les informations retournées par `wait` sont codées sur un entier court, selon le format visualisé sur la figure 13.1.

Si on exécute l'instruction

```
pid = wait(&statut);
```

la valeur de l'expression

```
statut & 0x7f
```

est le code d'erreur du programme fils et

```
statut >> 8
```

est sa valeur de retour. Lorsque le code d'erreur est différent de zéro, le bit numéro 7 est positionné si un fichier `core` a été produit.

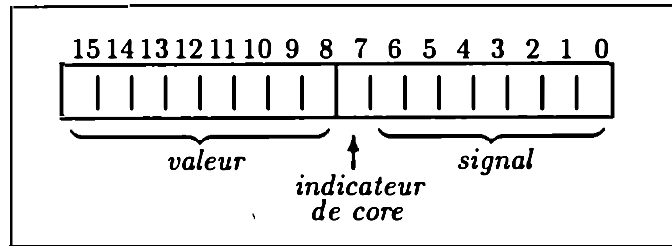


Figure 13.1: Statut de terminaison d'un processus

Le programme suivant, `printexit`, est un exemple de récupération de ces différentes informations.

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

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

    if (fork())
        wait(&statut);
    else
        execv(argv[1], argv+1);
    printf("%s termine avec \n", argv[1]);
    printf(" - la valeur %d\n", statut >> 8);
    printf(" - le statut %d\n", statut & 0x7f);
    printf(" - indic. de core %d\n", (statut & 0x80) != 0);
}
printexit.c
```

La session suivante enchaîne le lancement par `printexit`

- d'une commande se terminant normalement,
- d'une commande retournant un code de sortie non nul,
- d'une commande terminant sur exception avec production d'un fichier `core`,
- d'une commande terminant sur exception sans production d'un fichier `core`.

```
$ printexit /bin/date
Sun Aug 27 04:52:14 PDT 1989
/bin/date termine avec
 - la valeur 0
 - le statut 0
 - indic. de core 0
$
$ printexit exit_with 3
exit_with termine avec
 - la valeur 3
 - le statut 0
 - indic. de core 0
$
```

```
$ printexit autokill 3
3: autokill termine avec
 - la valeur 0
 - le statut 3
 - indic. de core 1
$
$ printexit autokill 9
15: autokill termine avec
 - la valeur 0
 - le statut 9
 - indic. de core 0
$
```

Le programme `autokill`, qui s'envoie le signal spécifié par son premier argument est présenté page 434. Le programme `exit_with` est présenté page 423.

13.1.2 Attente d'un signal

Un processus endormi sur un `wait` est réveillé lorsqu'il reçoit un signal; dans le cas où le signal ne provoque pas sa terminaison, celui-ci se rendort une fois le traitement du signal effectué. Voici un exemple utilisant les appels `wait` et `kill` dans lequel un processus crée trois processus fils et interagit avec eux.

```
fork_wait.c
#include <signal.h>
#include <stdio.h>

#include "hms.h"

static void reception(void);
static void entete(char *);

main()
{
    int p1, p2, p3;
    int status;
    int pid;
    int h, m, s;

    signal(SIGUSR1, reception);
    if ((p1=fork()) == 0)
    {
        /* Premier fils */
        sleep(2);
        entete("envoi signal\n");
        kill(getppid(), SIGUSR1);
        sleep(3);
        entete("exit\n");
        exit(1);
    }
    if ((p2=fork()) == 0)
    {
```

```
        /* Second fils */
        sleep(15);
        entete("exit\n");
        exit(2);
    }
    if ((p3=fork()) == 0)
    {
        /* Troisieme fils */
        sleep(10);
        entete("exit\n");
        exit(3);
    }
    entete("wait\n");
    for (;;)
    {
        pid = wait(&status);
        if (pid == -1)
            break;
        entete("arret");
        printf(" de %d avec %d comme status\n",
                pid, status>>8);
    }
}

static void reception()
{
    entete("signal recu\n");
}

static void entete(char *message)
{
    int h, m, s;

    hms(&h, &m, &s);
    printf("[%d] %02dh %02dm %02ds : %s",
            getpid(), h, m, s, message);
}
```

fork_wait.c

Chaque message affiché est préfixé par le numéro du processus qui en est l'auteur et par l'heure de l'émission. L'exécution du programme précédent se déroule ainsi :

```
$ fork_wait
[263] 00h 12m 40s : wait
[264] 00h 12m 42s : envoi signal
[263] 00h 12m 42s : signal recu
[264] 00h 12m 45s : exit
[263] 00h 12m 45s : arret de 264 avec 1 comme status
[266] 00h 12m 50s : exit
[263] 00h 12m 50s : arret de 266 avec 3 comme status
[265] 00h 12m 55s : exit
[263] 00h 12m 55s : arret de 265 avec 2 comme status
$
```

Les signaux peuvent être utilisés pour effectuer des synchronisations entre processus. Une possibilité consiste à endormir un processus au moyen de l'appel **PAUSE(2)**. Le processus ainsi endormi sera réveillé par la réception d'un signal.

L'exemple suivant montre un processus se dupliquant au moyen d'un `fork`. Le processus père et le processus fils alternent alors des séquences de la forme

> `printf(`*un message*`);`
> `kill(`*pid de l'autre processus*`, SIGUSR1);`
> `pause();`

Dans chaque processus, le signal `SIGUSR1` est associé à une fonction exécutant l'appel `sleep` afin de s'endormir durant une seconde.

avecpause.c

```
#include <signal.h>

void sigusr();
void pere(int);
void fils(int);

main()
{
    int pid;

    signal(SIGUSR1, sigusr);
    pid = fork();
    if (pid != 0)
        pere(pid);
    else
        fils(getppid());
}

void pere(int pid)
{
    printf("Pere: 1\n");
    kill(pid, SIGUSR1);
    pause();
    printf("Pere: 2\n");
    kill(pid, SIGUSR1);
    pause();
}

void fils(int pid)
{
    pause();
    printf("Fils: 1\n");
    kill(pid, SIGUSR1);
    pause();
    printf("Fils: 2\n");
    kill(pid, SIGUSR1);
}

void sigusr()
{
    sleep(1);
```

```
}
```

avecpause.c

Lors de l'exécution de la commande **avecpause**, les deux processus alternent leurs écritures :

```
$ avecpause
Pere: 1
Fils: 1
Pere: 2
Fils: 2
$
```

En revanche, la version du même programme écrite sans synchronisation produira les affichages selon un ordre indéterminé, dépendant de la gestion des processus par le système de temps partagé. La commande **sanspause** est obtenue en enlevant les instructions

```
pause();
```

du source de la commande **avecpause**. Voici trois exécutions différentes que nous avons obtenues sur un *SUN* sous *OS-4* :

```
$ sanspause
Pere: 1
Fils: 1
Fils: 2
Pere: 2
$
$ sanspause
Fils: 1
Pere: 1
Pere: 2
$ Fils: 2

$
$ sanspause
Fils: 1
Fils: 2
Pere: 1
Pere: 2
$
```

On remarque que lors de la seconde exécution, le processus père termine pendant que le processus fils est endormi, d'où l'affichage du *prompt shell* interférant avec le second message du fils.

Remarque 41 *Le mécanisme présenté ici peut provoquer des blocages. En effet, si on enlève l'appel à* `sleep` *de la fonction* `sigusr`*, il est possible que l'un des processus exécute l'envoi de signal avant que l'autre se soit endormi. Dans ce cas les deux processus se retrouvent "en pause" en même temps.*

13.2 Utilisation des tubes standard

Nous avons décrit en 10.2.1 (page 336) le principe de fonctionnement d'un tube d'entrées-sorties, ou *pipe*, et nous en avons donné un exemple page 344. L'intérêt des tubes est de permettre la communication entre deux processus différents. Cela se fait en enchaînant les deux appels système `pipe` et `fork`.

Le premier crée un tube de communication en allouant un vecteur de deux descripteurs d'entrées-sorties, et le second crée un processus fils qui hérite des entrées-sorties de son père. Les deux processus peuvent alors échanger des informations à travers le tube, puisqu'ils ont tous deux la possibilité de lire et d'écrire dedans.

Pour se dérouler correctement, une communication par tube doit être à sens unique : du père vers le fils, ou du fils vers le père. Si on ne respecte pas cette règle, les données issues du père et celles issues du fils se mêlent, et une donnée peut très bien être relue par le processus qui l'a écrite.

Par conséquent, à moins d'établir un protocole de communication très strict entre les processus communicants, ce qui n'est pas en général chose facile, il est naturel de créer autant de tubes qu'il y a de communications unilatérales.

Une première possibilité consiste à faire communiquer un processus père et un processus fils au moyen de deux tubes, un du père vers le fils et l'autre du fils vers le père. Soient `fdpf` le tube utilisé dans le sens père-fils, et `fdfp` l'autre :

```
int fdpf[2];
int fdfp[2];
```

L'initialisation des communications s'effectue selon le principe illustré par la figure 13.2.

```
/* Ouverture des tubes */
if (pipe (fdpf) == -1 || pipe (fdfp) == -1)
    gestion de l'erreur
else if (fork () == 0) {
    /* Processus fils */
    close (fdpf [1]);
    close (fdfp [0]);
    Traitements avec
        lectures des données sur fdpf [0]
        écritures résultats sur fdfp [1]
}
else {
    /* Processus pere */
    close (fdpf [0]);
    close (fdfp [1]);
    Entrées-sorties sur le terminal avec
        écritures des données sur fdpf [1]
        lectures des résultats sur fdfp [0]
}
```

Figure 13.2: Communication père-fils

Ce schéma ne pose pas de problème particulier. Pour éviter les risques de blocage, on prendra soin de refermer une communication dès qu'elle n'est plus utilisée. Ainsi, la détection de fin de fichier s'effectuera correctement. On pourra également récupérer le signal SIGPIPE envoyé par le système à un processus qui écrit dans un tube dont toutes les entrées ont été fermées.

Voici un exemple simple illustrant ces divers mécanismes. Le processus père associe le signal SIGPIPE à la fonction **sigpipe**, qui imprime un message et provoque la terminaison du programme. Il enchaîne ensuite l'écriture d'un **'A'** dans le tube *père-fils*, la lecture d'un caractère dans le tube *fils-père* et son affichage à l'écran, la recopie de ce même caractère dans le tube *père-fils*, et enfin la lecture d'un dernier caractère dans le tube *fils-père*. Ce traitement est décrit dans la fonction **pere**.

Le processus fils lit un caractère dans le tube père-fils et l'affiche à l'écran, incrémente ce caractère de un et le recopie dans le tube fils-père. Ce traitement est décrit dans la fonction **fils**.

pere_fils.c

```
#include <stdio.h>

#include <signal.h>
void sigpipe();

int fdpf[2];
int fdfp[2];

void sigpipe();
void pere();
void fils();

main()
{
    if (pipe(fdfp) == -1 || pipe(fdpf) == -1)
        exit(1);
    if (fork() == 0)
    {
        close(fdpf[1]);
        close(fdfp[0]);
        fils();
    }
    else
    {
        close(fdpf[0]);
        close(fdfp[1]);
        pere();
    }
    printf("----(%d): arret\n", getpid());
}

void sigpipe()
{
    printf("Pere(%d): recu signal SIGPIPE - arret\n",
           getpid());
    exit(0);
}
```

```
void pere()
{
    char c = 'A';

    signal(SIGPIPE, sigpipe);
    write(fdpf[1], &c, 1);
    read(fdfp[0], &c, 1);
    printf("Pere(%d): recu '%c' dans le tube\n",
           getpid(), c);
    write(fdpf[1], &c, 1);
    read(fdfp[0], &c, 1);
    printf("Pere(%d): recu '%c' dans le tube\n",
           getpid(), c);
}

void fils()
{
    char c;

    read(fdpf[0], &c, 1);
    printf("Fils(%d): recu '%c' dans le tube\n",
           getpid(), c);
    c++;
    write(fdfp[1], &c, 1);
}
```

pere_fils.c

L'exécution du programme se déroule de la façon suivante :

```
$ tube
Fils(4209): recu 'A' dans le tube
----(4209): arret
Pere(4208): recu 'B' dans le tube
Pere(4208): recu signal SIGPIPE - arret
$
```

La seconde écriture dans le tube *père-fils* par le processus père provoque l'envoi d'un signal SIGPIPE, car il n'y a plus d'entrée active sur le tube.

Un schéma un peu plus complexe est celui décrit sur la figure 13.3. Il s'agit de faire communiquer deux processus fils entre eux, de façon symétrique. Cette fois encore on utilise deux tubes, le tube 1 $\rightarrow$ 2 du premier fils vers le second et le tube 2 $\rightarrow$ 1 pour la communication inverse. Le processus père crée successivement deux fils puis attend leur terminaison après avoir refermé ses entrées-sorties sur les tubes. Pour cela, il enchaîne les actions suivantes :

1) création des deux tubes

```
pipe(fd12);
pipe(fd21);
```

si l'une des deux créations échoue, sortie sur erreur;

2) création d'un premier fils *pid1*

```
pid1 = fork();
```

3) création d'un second fils *pid2*

```
pid2 = fork();
```

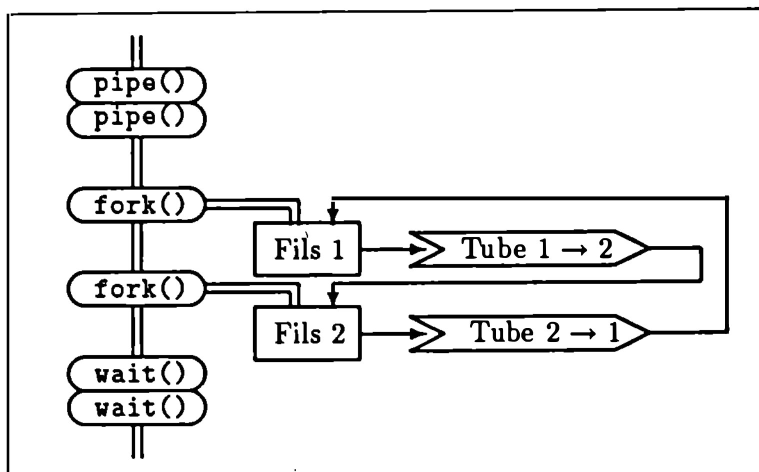


Figure 13.3: Communication entre deux processus fils

si l'une des deux duplications échoue, sortie sur erreur (si c'est la seconde, envoie d'un signal `SIGKILL` au premier fils)

4) fermeture des entrées-sorties sur les tubes (le père ne communique pas avec ses fils).

```
close(fd12[0]);
close(fd12[1]);
close(fd21[0]);
close(fd21[1]);
```

5) attente de la terminaison des deux fils : boucle itérant

```
pidexit = wait(0);
```

tant que `pidexit` n'a pas pris les deux valeurs *pid1* et *pid2*, dans un ordre quelconque.

Chaque fils commence par fermer les entrées-sorties qu'il n'utilise pas, puis déroule son propre code. Cela donne pour le premier :

```
close(fd12[0]);
close(fd21[1]);
```
Code du fils numéro 1

et pour le second :

```
close(fd21[0]);
close(fd12[1]);
```
Code du fils numéro 2

Les deux processus fils peuvent maintenant travailler en parallèle, tandis que leur père attend leur terminaison.

Nous allons illustrer cette construction au moyen de la commande **p1p2** qui reçoit en arguments les noms des deux commandes à lancer, initialise les tubes comme nous venons de le décrire, et lance chaque commande en lui fournissant en argument le descripteur sur lequel elle doit lire et celui sur lequel elle doit écrire. Par exemple, si les descripteurs de fichiers associés aux tubes sont 3, 4 pour le tube 1 → 2 et 5, 6 pour le tube 2 → 1, et qu'on fasse exécuter **p1p2 p1 p2** le premier fils de **p1p2** fera exécuter **p1 5 4** et le second **p2 3 6** (voir

figure 13.4).

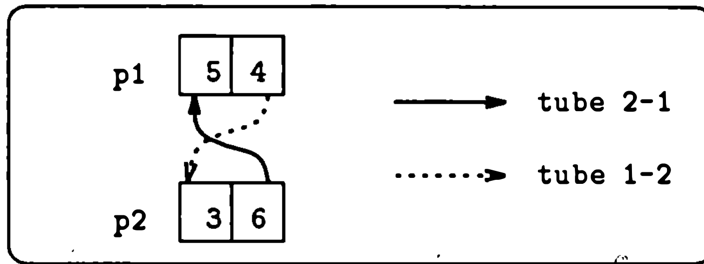


Figure 13.4: Entrées-sorties initialisées par **p1p2**

Le source de cette commande est le suivant :

```
═══════ p1p2.c ═══════
#include <stdio.h>
#include <signal.h>
#define TCOM 1024

int fd12[2];
int fd21[2];
int pid1;
int pid2;
char com[TCOM];
void pere();
void fils1(char *);
void fils2(char *);

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

    if (pipe(fd12) == -1 || pipe(fd21) == -1)
        exit(1);
    if ((pid1=fork()) == -1)
        exit(1);
    if (pid1 > 0)
    {
        if ((pid2=fork()) == -1)
        {
            kill(pid1, SIGKILL);
            exit(1);
        }
        if (pid2 > 0)
            pere();
        else
            fils2(argv[2]);
    }
    else
        fils1(argv[1]);
}

void pere()
{
    int nb = 2;
```

```
    close(fd12[0]);
    close(fd12[1]);
    close(fd21[0]);
    close(fd21[1]);
    while (nb)
    {
        int pidexit = wait(0);

        if (pidexit == pid1)
            nb--;
        else if (pidexit == pid2)
            nb--;
    }
}

void fils1(char *s)
{
    close(fd12[0]);
    close(fd21[1]);
    sprintf(com, "%s %d %d", s, fd21[0], fd12[1]);
    printf("Lancement de \"%s\"\n", com);
    execl("/bin/sh", "sh", "-c", com, 0);
}

void fils2(char *s)
{
    close(fd12[1]);
    close(fd21[0]);
    sprintf(com, "%s %d %d", s, fd12[0], fd21[1]);
    printf("Lancement de \"%s\"\n", com);
    execl("/bin/sh", "sh", "-c", com, 0);
}
```
p1p2.c

La terminaison des fils peut poser des problèmes de blocage dus à la situation symétrique des deux processus. Une solution consiste à les faire communiquer par signal : un des processus (voire les deux) peut commencer par envoyer son numéro d'identification dans le tube; son frère sera alors en mesure de lui envoyer un signal.

Voici un exemple comportant deux commandes `p1` et `p2`, dans lequel la terminaison est gérée par le processus `p1`. Pour cela, `p2` écrit son `pid` dans le tube 2 → 1.

L'exécution du processus `p1` commence par la lecture de ce `pid` dans le tube. Il initialise ensuite le traitement en lisant un entier sur son entrée standard, la variable `borne`, et en écrivant successivement 0 et 1 dans le tube 1 → 2. Il enchaîne ensuite la lecture d'un nombre lu sur le tube 2 → 1 et sa recopie à l'écran et dans le tube 1 → 2. Le traitement s'arrête lorsque le nombre lu est supérieur à la valeur de la variable `borne`.

p1.c
```
#include <stdio.h>
#include <signal.h>

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

```
    int fdi = atoi(argv[1]);
    int fdo = atoi(argv[2]);
    unsigned long n = 0;
    int pid;
    int borne;

    read(fdi, &pid, sizeof pid);
    printf("Borne: ");
    scanf("%d", &borne);
    write(fdo, &n, sizeof n);
    n++;
    write(fdo, &n, sizeof n);

    for (;;)
    {
        read(fdi, &n, sizeof n);
        if (n >= borne)
            break;
        printf("%d ", n);
        write(fdo, &n, sizeof n);
    }
    printf("\n");
    kill(pid, SIGUSR1);
    exit(0);
}
```

p1.c

Le processus p2 récupère le signal SIGUSR1 et écrit son *pid* dans le tube $2 \rightarrow 1$. Il itère

- la lecture d'un entier i dans le tube $1 \rightarrow 2$ qu'il utilise pour incrémenter une variable n;
- la recopie de n dans le tube $2 \rightarrow 1$.

p2.c

```
#include <signal.h>

main(int argc, char *argv[])
{
    int fdi = atoi(argv[1]);
    int fdo = atoi(argv[2]);
    int pid = getpid();
    unsigned long n;
    void sigusr1();

    signal(SIGUSR1, sigusr1);
    write(fdo, &pid, sizeof pid);
    read(fdi, &n, sizeof n);
    for (;;)
    {
        unsigned long i;

        write(fdo, &n, sizeof n);
        if (read(fdi, &i, sizeof i) == 0)
            break;
        n += i;
    }
```

```
}

void sigusr1()
{
    exit(0);
}
```
p2.c

Nous laissons au lecteur le soin d'analyser cet algorithme dont voici un exemple d'exécution :

```
$ pip2 p1 p2
Lancement de "p1 5 4"
Lancement de "p1 3 6"
Borne: 4000
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584
$
```

On peut bien évidemment imaginer toutes sortes de schémas de communication : en étoile entre un père et ses fils, en anneau entre plusieurs fils... Le principe de mise en œuvre reste le même.

13.3 Redirection d'entrée-sortie dans un tube

Dans la section précédente, nous avons vu comment faire communiquer ensemble des processus ayant été prévus pour cette communication. Nous allons maintenant aborder un mécanisme différent consistant à détourner l'entrée et/ou la sortie standard d'un processus dans un tube. Il s'agit par conséquent d'une redirection d'entrées-sorties standard, comparable à la redirection dans un fichier présentée en 12.3.3.

Dans le cas de redirection dans un fichier, il suffit d'enchaîner la fermeture de l'entrée-sortie à redéfinir avec une ouverture du fichier sur lequel s'effectue la redirection. Dans le cas des tubes ce mécanisme ne convient pas. En effet, la suite d'instructions nécessaires à cette mise en œuvre

```
close(0);
close(1);
pipe(fd);
if (fork())
{
    ...
```

aurait pour résultat d'effectuer la redirection dans le processus père également.

La configuration qu'on veut créer est celle visualisée sur la figure 13.5. Elle se compose de deux tubes, $p \rightarrow f$ du père vers le fils et $f \rightarrow p$ du fils vers le père, avec :

- pour le processus père :
  * 0 associé à l'entrée standard
  * 1 associé à la sortie standard
  * 2 associé à la sortie d'erreur standard

    * `fdfp[0]` associé à la sortie du tube $f \rightarrow p$
    * `fdpf[1]` associé à l'entrée du tube $p \rightarrow f$
- pour le processus fils :
    * `0` associé à la sortie du tube $p \rightarrow f$
    * `1` associé à l'entrée du tube $f \rightarrow p$
    * `2` associé à la sortie d'erreur standard

les autres descripteurs étant fermés. Ce schéma est décrit sur la figure 13.5.

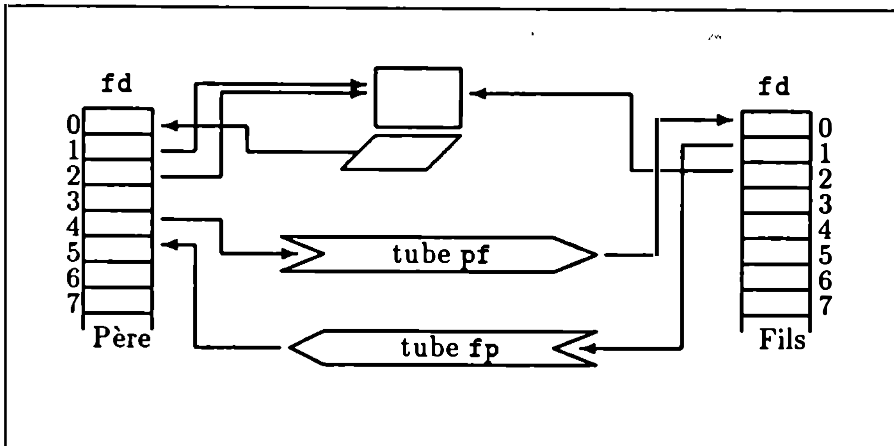


Figure 13.5: Détournement d'entrées-sorties dans un tube

Le père peut continuer à effectuer des entrées-sorties au terminal, et parallèlement à communiquer par les tubes avec son fils. La puissance de ce mécanisme réside dans le fait que le processus fils peut se substituer à un programme exécutable quelconque, programme utilisateur ou commande du système; le nouvel exécutable effectuera ses entrées-sorties standard dans les tubes.

Nous allons maintenant détailler les différentes étapes de mise en œuvre de ce mécanisme. La figure 13.6 montre l'environnement d'entrées-sorties du processus père après avoir créé les deux tubes (on suppose qu'avant de créer les communications, le processus père n'a pas initialisé d'autre entrée-sortie).

Le processus père exécute ensuite l'appel `fork` et le fils ainsi créé hérite des entrées-sorties de son père. La phase suivante consiste à fermer les entrées-sorties inutiles. Le père (resp. le fils) n'a plus besoin de l'accès en lecture dans le tube qu'il utilise pour écrire à son fils (resp. père). Le père (resp. le fils) ferme par conséquent le descripteur `fdpf[0]` (resp. `fdfp[0]`). De façon symétrique, on ferme les descripteurs associés à des sorties inutilisées. Enfin, le processus fils libère ses entrées-sorties standard pour préparer la redirection. La figure 13.7 montre l'environnement d'entrées-sorties des deux processus après le `fork`; les canaux destinés à être fermés sont marqués d'un point noir.

La troisième étape consiste à associer l'entrée standard du fils au canal de lecture du tube $p \rightarrow f$, et sa sortie standard au canal d'écriture du tube $f \rightarrow p$. Il s'agit de dupliquer les canaux marqués d'un point sur la figure 13.8, afin d'obtenir ceux représentés par des flèches obliques.

Cette opération de duplication est possible grâce à l'appel **DUP(2)** qui accepte en paramètre un descripteur correctement initialisé et le duplique sur la première entrée libre du tableau des descripteurs. Il suffit donc d'exécuter

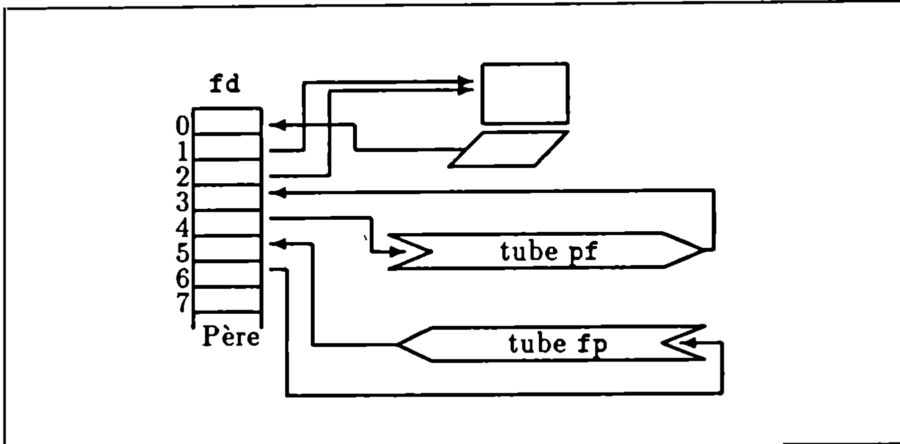


Figure 13.6: Pipecmd : initialisation de deux tubes

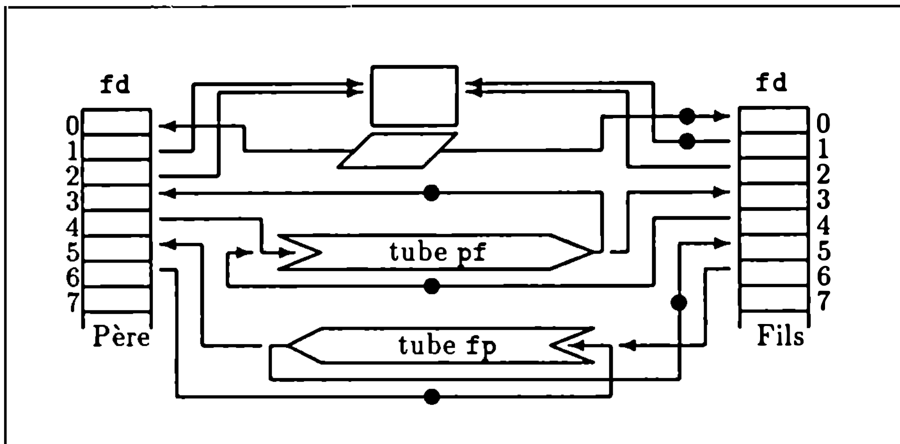


Figure 13.7: Pipecmd : fermeture des canaux inutiles

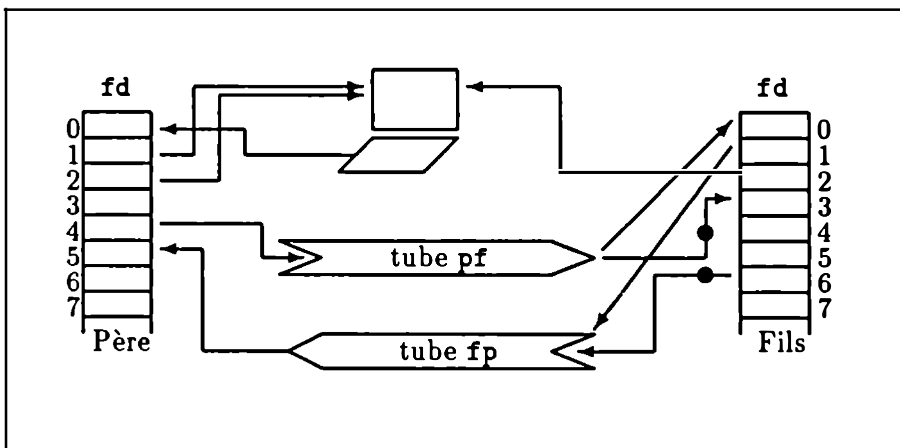


Figure 13.8: Pipecmd : duplication des canaux de tubes du fils.

```
close(0);
dup(fdpf[0]);
```

pour effectuer la première redirection, et

```
close(1);
dup(fdfp[1]);
```

pour la seconde.

Il ne reste plus, dans une dernière étape, qu'à fermer dans le processus fils les deux canaux devenus inutiles (canaux marqués d'un point sur la figure 13.8). On remarquera que la sortie d'erreur standard du processus n'a pas été modifiée. D'autre part, chaque tube est connecté à une seule entrée et une seule sortie. Il est maintenant possible au fils de se substituer par un **exec** à commande quelconque pour que son père puisse dialoguer avec elle.

Il est remarquable que la mise en œuvre d'une opération à priori complexe soit somme toute aussi simple. En effet, le code de la fonction `pipecmd` effectuant les redirections que nous venons de décrire et le lancement d'une commande, tient en quelques lignes. Cette fonction reçoit en paramètre la commande à lancer; elle retourne l'identificateur du processus créé dans le cas ou le lancement se déroule correctement et zéro sinon.

Il est très facile d'écrire un ensemble de fonctions utilisables par le processus père pour communiquer avec son fils. Par exemple `pipeputs` qui écrit une chaîne de caractères dans le tube en la terminant par un retour chariot, `pipegets` qui effectue l'opération symétrique, et `pipeputc` et `pipegetc` effectuant l'écriture et la lecture d'un caractère. Enfin, les fonctions `pipecloseout`, `pipeclosein` et `pipeclose` permettent de fermer respectivement, le canal d'écriture, le canal de lecture, et les deux canaux simultanément.

```
============ pipecmd.c ============================================
#include <stdio.h>
#include "pipecmd.h"

static int fdpf[2];
static int fdfp[2];

/* Lancement d'une commande redirigee dans deux tubes */
int pipecmd(char *cmd)
{
    int pid;

    if (pipe(fdpf) == -1 || pipe(fdfp) == -1)
        return 0;
    pid = fork();
    switch (pid)
    {
      case 0:
        /* fils: 0 --> entree pf, 1 --> sortie fp */
        close(0);
        dup(fdpf[0]);
        close(1);
        dup(fdfp[1]);
        /* Fermeture des descripteurs inutilises */
        close(fdpf[0]);
        close(fdpf[1]);
        close(fdfp[0]);
        close(fdfp[1]);
```

```
        /* Lancement de la commande  */
        execl("/bin/sh", "(sh)", "-c", cmd, 0);
        perror("/bin/sh");
        exit(1);
        /*NOBREAK*/

      case -1:
        perror("pipecmd");
        return 0;
        /*NOBREAK*/

      default:
        /* pere: fdpf[1] et fdfp[0] restent ouverts */
        close(fdpf[0]);
        close(fdfp[1]);
        return pid;
        /*NOBREAK*/
    }
}

/* Retourne le nombre de caracteres ecrits */
int pipeputs(char *s)
{
    int n = write(fdpf[1], s, strlen(s));

    return n + write(fdpf[1], "\n", 1);
}

/* Retourne la longueur de la chaine; 0 si fin de fichier */
int pipegets(char *s, int l)
{
    char *p = s;

    while (--l)
    {
        if (read(fdfp[0], p, 1) < 1)
            break;
        if (*p == '\n')
            break;
        p++;
    }
    *p = '\0';
    return p-s;
}

/* Retourne 1 si ecriture ok, 0 sinon */
int pipeputc(char c)
{
    return write(fdpf[1], &c, 1);
}

/* Retourne un caractere ou EOF */
int pipegetc()
{
    char c;

    if (read(fdfp[0], &c, 1) <= 0)
        return EOF;
```

```
    return c;
}

/* Fonctions de fermeture */
void pipecloseout()
{
    close(fdpf[1]);
}

void pipeclosein()
{
    close(fdfp[0]);
}

void pipeclose()
{
    pipeclosein();
    pipecloseout();
}
```
════════════════ pipecmd.c ════════

Le module pipecmd que nous venons de détailler permet à un programme de disposer de toute la puissance de l'environnement de programmation UNIX. L'exemple suivant montre l'écriture d'un mini-calculateur utilisant le calculateur multi-précision DC(1), qui évalue des expressions en notation postfixe. Par exemple, la suite de commandes `22 3 +p` envoyée à dc génère successivement

- l'empilement de 22;
- l'empilement de 3;
- le dépilement de ces deux valeurs suivi du calcul de leur somme;
- l'empilement du résultat;
- l'impression du sommet de pile (commande p).

════════ calc.c ════════════════
```
#include <stdio.h>
#include "pipecmd.h"
#include "affiche_cars.h"

static int max(int, int);
static void afficher(char, char *, int);

main(int argc, char *argv[])
{
    int pid;
    int l;
    char resultat[1024];

    if (argc != 4)
    {
        fprintf(stderr, "Usage %s  <nb> <op> <nb>\n", *argv);
        exit(1);
    }
    pid = pipecmd("dc");
    if (pid == 0)
```

```
        exit(1);

    l = pipeputs(argv[1]);
    l = max(l, pipeputs(argv[3]));
    pipeputc(argv[2][0]);
    pipeputc('p');
    pipecloseout();
    l = max(l, pipegets(resultat, sizeof resultat));
    pipeclosein();

    afficher(' ', argv[1], l);
    afficher(argv[2][0], argv[3], l);
    printf("  ");
    affiche_cars('-', l);
    printf("\n");
    afficher('=', resultat, l);
}

static void afficher(char c, char *s, int l)
{
    printf("%c %*s\n",  c, l, s);
}

static int max(int a, int b)
{
    return a > b ? a : b;
}
```
calc.c

Pour calculer une multiplication, il est nécessaire de *quoter* le caractère * afin qu'il ne soit pas interprété par le shell.

```
$ calc 1213141516171819 - 919293949596979899
    1213141516171819
-  919293949596979899
  ------------------
= -918080808080808080
$
$ calc 99999999999999999999 \* 101010101010101010
                     99999999999999999999
*                      101010101010101010
  ---------------------------------------
= 1010101010101010099998989898989898990
$
```

La terminaison de l'exécution de dc est provoquée par la fermeture du tube sur lequel son entrée standard est redirigée, ce qui entraine une détection de fin de fichier.

Annexe A

Mémento des définitions standard

Cette annexe est un aide-mémoire résumant les principales définitions de l'environnement de programmation C tel qu'il a été spécifié dans la norme ANSI. Elle s'appuie principalement sur deux documents : la description de la bibliothèque donnée dans *"The C Programming Language"* de Brian W.Kernighan et Dennis M.Ritchie, et le document *"Rationale for American National Standard for Information Systems – Programming Language – C"* produit par le comité de normalisation X3J11[1].

Les définitions de l'environnement standard sont, soit des définitions de types, de variables, de constantes et de pseudo-fonctions contenues dans l'un des quinze fichiers en-tête standard, soit des fonctions contenues dans la bibliothèque standard. Sous UNIX, les définitions de fonction sont traditionnellement réparties dans deux fichiers : `libc.a` contenant les définitions des fonctions de la bibliothèque standard sauf les fonctions mathématiques, et `libm.a` contenant les définitions des fonctions mathématiques. Par défaut, seule la première est consultée. La seconde doit être explicitement demandée au moyen de l'option d'édition de liens `-lm`.

A.1 Principales définitions de type

Ces types ont été définis pour des raisons de portabilité, afin de permettre l'écriture de programmes indépendants de l'implémentation. Chacune des lignes suivantes décrit sommairement un type, en donnant son nom, une description de l'objet qu'il déclare, et le fichier en-tête à inclure pour l'utiliser.

`clock_t`	: temps CPU (unité propre à l'implémentation)	`<time.h>`
`div_t`	: résultat division entière (`struct{quot,rem}`)	`<stdlib.h>`
`fpos_t`	: positionnement dans un flot	`<stdio.h>`
`jmp_buf`	: tampon de sauvegarde de contexte	`<setjmp.h>`
`ldiv_t`	: comme `div_t` avec des entiers longs	`<stdlib.h>`
`ptrdiff_t`	: différence de deux pointeurs	`<stddef.h>`
`size_t`	: taille d'un objet (type de `sizeof`)	`<stddef.h>`

[1]Ce document est disponible sur le réseau `ftp`, par *ftp anonyme* sur `ftp.uu.net`, dans le répertoire `/doc/standards/ansi/X3.159-1989`.

sig_atomic_t	: variable volatile de handler de signal	<signal.h>
struct lconv	: conventions numériques locales	<locale.h>
struct tm	: temps externe (struc{tm_year,tm_hour...})	<time.h>
time_t	: temps interne de la machine	<time.h>
va_list	: liste de paramètres variables	<stdarg.h>
wchar_t	: caractères étendus	<stddef.h>

A.2 Principales définitions de constantes

Ces constantes sont en général des pseudo-constantes définies par des directives au préprocesseur. Elles décrivent principalement, des limites de l'implémentation, des paramètres de fonctions de la bibliothèque, et des constantes de conversion. Chacune des lignes suivantes décrit sommairement une constante, en donnant son nom, sa description sommaire, et le fichier en-tête à inclure pour l'utiliser.

__DATE__ (char*)	: date de la compilation	
__FILE__ (char*)	: fichier source courant	
__LINE__	: numéro de la ligne courante	
__STDC__	: compilateur norme ANSI	
__TIME__ (char*)	: heure de la compilation	
_IOFBF	: E/S mode mémorisation complète	<stdio.h>
_IOLBF	: E/S mode mémorisation par ligne	<stdio.h>
_IONBF	: E/S mode non mémorisé	<stdio.h>
BUFSIZ	: taille d'un tampon d'E/S	<stdio.h>
CHAR_BIT	: nombre de bits dans un caractère (octet).	<limits.h>
CHAR_MAX	: valeur maximale du type char	<limits.h>
CHAR_MIN	: valeur minimale du type char	<limits.h>
CLOCKS_PER_SEC	: nombre d'unités de temps CPU par seconde	<time.h>
DBL_DIG	: précision décimale d'un double	<float.h>
DBL_EPSILON	: précision minimale d'un double	<float.h>
DBL_MANT_DIG	: précision décimale de la mantisse	<float.h>
DBL_MAX_EXP	: précision maximale de l'exposant	<float.h>
DBL_MAX	: précision maximale d'un double	<float.h>
DBL_MIN_EXP	: précision minimale de l'exposant	<float.h>
DBL_MIN	: précision minimale d'un double	<float.h>
EDOM	: paramètre hors domaine de définition (math)	<errno.h>
ERANGE	: résultat de fonction inexprimable (math)	<errno.h>
EOF	: constante fin de fichier	<stdio.h>
FILENAME_MAX	: taille maximum d'un nom de fichier	<stdio.h>
FLT_DIG	: précision décimale d'un float	<float.h>
FLT_EPSILON	: précision minimale d'un float	<float.h>
FLT_MANT_DIG	: précision décimale de la mantisse	<float.h>
FLT_MAX_EXP	: précision maximale de l'exposant	<float.h>
FLT_MAX	: précision maximale d'un float	<float.h>
FLT_MIN_EXP	: précision minimale de l'exposant	<float.h>
FLT_MIN	: précision minimale d'un float	<float.h>
FLT_RADIX	: représentation de l'exposant d'un float	<float.h>
FLT_ROUNDS	: mode de calcul d'arrondi	<float.h>
FOPEN_MAX	: maximum de flots ouvrables simultanément	<stdio.h>
HUGE_VAL	: plus grande valeur codable ($+\infty$)	<math.h>
INT_MAX	: valeur maximale du type signed int	<limits.h>

INT_MIN	: valeur minimale du type signed int	<limits.h>
LC_ALL	: toutes les catégories de paramètres locaux	<locale.h>
LC_COLLATE	: paramétrage de strcoll et strxfrm	<locale.h>
LC_CTYPE	: paramétrage local de ctype	<locale.h>
LC_MESSAGES	: paramétrage des messages locaux	<locale.h>
LC_MONETARY	: paramétrage local des quantités monétaires	<locale.h>
LC_NUMERIC	: paramétrage local du formatage des nombres	<locale.h>
LC_TIME	: paramétrage du format du temps externe	<locale.h>
LONG_MAX	: valeur maximale du type signed long int	<limits.h>
LONG_MIN	: valeur minimale du type signed long int	<limits.h>
L_tmpnam	: longueur maximale nom de fichier temporaire	<stdio.h>
MB_LEN_MAX	: taille maximale d'un multi-caractère	<limits.h>
NDEBUG	: contrôle du mécanisme d'assertion	<assert.h>
NULL	: pointeur indéfini (valeur 0)	<stddef.h>
RAND_MAX	: borne supérieure ($\geq$ 32767) de la fonction rand	<stdlib.h>
SCHAR_MAX	: valeur maximale du type signed char	<limits.h>
SCHAR_MIN	: valeur minimale du type signed char	<limits.h>
SEEK_CUR	: positionnement relatif (fseek)	<stdio.h>
SEEK_END	: positionnement absolu inverse (fseek)	<stdio.h>
SEEK_SET	: positionnement absolu (fseek)	<stdio.h>
SHRT_MAX	: valeur maximale du type signed short int	<limits.h>
SHRT_MIN	: valeur minimale du type signed short int	<limits.h>
SIGABRT	: signal provoquant une terminaison anormale	<signal.h>
SIGFPE	: signal émis par une erreur arithmétique	<signal.h>
SIGILL	: signal émis par sur une instruction illégale	<signal.h>
SIGINT	: signal émis interactivement depuis la console	<signal.h>
SIGSEGV	: signal émis par une erreur d'adressage	<signal.h>
SIGTERM	: signal de requête de terminaison	<signal.h>
SIG_DFL	: signal à initialiser (fonction signal)	<signal.h>
SIG_IGN	: signal à ignorer (fonction signal)	<signal.h>
TMP_MAX	: nombre maximum de noms fichiers temporaires	<stdio.h>
UCHAR_MAX	: valeur maximale du type unsigned char	<limits.h>
UINT_MAX	: valeur maximale du type nsigned int	<limits.h>
ULONG_MAX	: valeur maximale du type unsigned long int	<limits.h>
USHRT_MAX	: valeur maximale du type unsigned short int	<limits.h>

Remarque 42 *Il n'existe pas de définition des constantes* UCHAR_MIN, UINT_MIN, ULONG_MIN, *et* USHRT_MIN, *car elle correspondent à des valeurs nulles.*

A.3 Variables globales standard

Les quelques variables qui suivent sont des variables globales, dont la valeur est initialisée et/ou modifiée par l'environnement. Chacune des lignes suivantes décrit sommairement une variable, en donnant son nom, sa description sommaire, le fichier en-tête à inclure pour l'utiliser.

int	errno	: code d'erreur des fonctions standard	<errno.h>
FILE*	stderr	: flot de sortie standard erreur	<stdio.h>
FILE*	stdin	: flot d'entrée standard	<stdio.h>
FILE*	stdout	: flot de sortie standard	<stdio.h>

A.4 Fonctions et pseudo-fonctions standard

Les définitions qui suivent peuvent être des pseudo-fonctions définies à l'aide de directives du préprocesseur ou des fonctions véritable. Dans le premier cas, elles ne correspondent pas à des symboles globaux de l'environnement et ne peuvent être utilisés en tant que tel, par exemple, pour être passées en paramètre d'une autre fonction. Les définitions qui sont explicitement des macro-fonctions sont signalées par un (MACRO) au début de leur description. Pour chaque fonction décrite, on trouvera son prototype, le fichier ou les fichiers en-tête à inclure pour l'utiliser, et sa description sommaire.

Les descriptions sont volontairement limitées à quelques lignes afin de ne pas dupliquer les informations développées dans le reste de l'ouvrage, comme les spécifications de format, les structures décrivant des parametres (tm)...

Les prototypes donnent simplement les types des paramètres, sans faire apparaître d'identificateur. Dans la description d'une fonction avec un seul paramètre formel, ce paramètre sera noté p. Lorsqu'il y a plusieurs paramètres formels, il seront notés respectivement p_1, p_2, p_3...; dans ce cas, p_1 désigne le premier paramètre, p_2 le second, et ainsi de suite.

void ____ abort(void) ____ <stdlib.h>

Cette fonction provoque la terminaison anormale du programme en envoyant le signal SIGABRT (également appelé SIGIOT sous UNIX). Attention, sous certaines implémentations non standard, lorsque ce signal est récupéré ou ignoré, la fonction abort est inopérante. [voir aussi signal,assert]

int ____ abs(int) ____ <stdlib.h>

Cette fonction retourne $|p|$, la valeur absolue de p. [voir aussi labs]

double ____ acos(double) ____ <math.h>

Cette fonction retourne *Arccos* p, la valeur de p devant être comprise dans l'intervalle $[-1, 1]$.

Erreur : HUGEVAL *et* errno=ERANGE *ou* errno=EDOM

char* ____ asctime(const struct tm*) ____ <time.h>

Cette fonction retourne une chaîne de caractères décrivant le temps dans le format externe (jour, mois, date, heures, minutes, secondes et année), construite à partir du temps local p. Cette chaîne est de la forme "Jou Moi da hh:mm:ss aaaa\n\0". [voir aussi localtime,strftime,time]

double ____ asin(double) ____ <math.h>

Cette fonction retourne *Arcsin* p, la valeur de p devant être comprise dans l'intervalle $[-1, 1]$.

Erreur : HUGEVAL *et* errno=ERANGE *ou* errno=EDOM

void ____ assert(int) ____ <assert.h>

(MACRO) Cette fonction affiche un message d'erreur sur la sortie erreur standard et effectue un appel à abort, lorsque p vaux 0 (*faux*). Le message d'erreur contient le nom du fichier et le numéro de ligne courants, et le texte de l'expression p. Désactivé par positionnement de la pseudo-constante NDEBUG. [voir aussi abort]

double ____ atan2(double, double) ____ <math.h>

Cette fonction retourne *Arctan* $\frac{p_1}{p_2}$, ou encore l'angle du nombre complexe $p_1 + ip_2$; les paramètres p_1 et p_2 ne peuvent être tous les deux nuls.

Erreur : HUGEVAL *et* errno=ERANGE *ou* errno=EDOM

double — atan(double) — <math.h>

Cette fonction retourne ***Arctan p***.
Erreur : HUGEVAL *et* errno=ERANGE *ou* errno=EDOM

int — atexit(void (*)(void)) — <stdlib.h>

Cette fonction installe la fonction pointée par p comme fonction de terminaison du programme, appelée automatiquement lors d'une exécution de exit. Il est possible d'empiler plusieurs fonctions qui seront successivement dépilées et exécutées lors de la terminaison. Lorsqu'une telle fonction exécute un longjmp, son comportement est indéterminé.
Erreur : $v \neq 0$ [voir aussi exit]

double — atof(const char*) — <stdlib.h>

Cette fonction retourne la valeur de la chaîne numérique réelle p, en défilant, s'il y a lieu, les *espaces* (voir fonction isspace) situés en tête.
Erreur : errno = ERANGE [voir aussi strtod]

int — atoi(const char*) — <stdlib.h>

Cette fonction retourne la valeur de la chaîne numérique entière décimale p, en défilant, s'il y a lieu, les *espaces* (voir fonction isspace) situés en tête.
Erreur : errno = ERANGE [voir aussi strtol]

long — atol(const char*) — <stdlib.h>

Cette fonction effectue le même calcul que atoi, mais avec un résultat long.
Erreur : errno = ERANGE [voir aussi strtol]

void * — bsearch(const void*, const void*, size_t, size_t, int(*)(const void*, const void*)) — <stdlib.h>

Cette fonction recherche par dichotomie la clé p_1 dans le vecteur p_2 composé de p_3 éléments de taille p_4, trié par ordre croissant pour la relation d'ordre implémentée par la fonction p_5 (cette fonction respectant les conventions de strcmp); elle retourne le pointeur vers l'élément trouvé ou NULL.
Erreur : NULL [voir aussi qsort,strcmp]

void* — calloc(size_t, size_t) — <stdlib.h>

Cette fonction retourne l'adresse d'une zone mémoire de taille $t = p_1 \times p_2$ (p_1 représente un nombre d'objets et p_2 la taille de ces objets), allouée dynamiquement et initialisée à zéro; elle retourne NULL si $t = 0$, ou sur erreur.
Erreur : NULL [voir aussi malloc,realloc,free]

double — ceil(double) — <math.h>

Cette fonction retourne le réel de valeur $\lceil p \rceil$ (le plus petit entier supérieur ou égal à p).
Erreur : HUGEVAL *et* errno=ERANGE *ou* errno=EDOM [voir aussi floor]

void — clearerr(FILE*) — <stdio.h>

Cette fonction réinitialise les indicateurs de fin de fichier et d'erreur du flot p.
[voir aussi feof,ferror]

clock_t — clock(void) — <time.h>

Cette fonction retourne le temps CPU écoulé depuis le lancement du programme. L'unité de temps est spécifique à chaque implémentation.
Erreur : -1 [voir aussi constante CLOCKS_PER_SEC]

double________ cos(double) ________ <math.h>

Cette fonction retourne ***cos p***, le cosinus de l'angle p exprimé en radiant.
Erreur : HUGEVAL *et* errno=ERANGE *ou* errno=EDOM

double________ cosh(double) ________ <math.h>

Cette fonction retourne ***cosh p***, le cosinus hyperbolique p.
Erreur : HUGEVAL *et* errno=ERANGE *ou* errno=EDOM

char*________ ctime(const time_t*) ________ <time.h>

Cette fonction retourne une chaîne de caractères décrivant le temps externe (jour, mois, date, heures, minutes, secondes et année), construite à partir du temps interne p. La chaîne construite est de la forme "Jou Moi da hh:mm:ss aaaa\n\0".
[voir aussi asctime,strftime,time]

double________ difftime(time_t, time_t) ________ <time.h>

Cette fonction retourne la différence exprimée en secondes entre les deux valeurs de temps interne p_1 et p_2. [voir aussi time,localtime,mktime]

div_t________ div(int, int) ________ <stdlib.h>

Cette fonction retourne le résultat de la division entière $p_1 \div p_2$; il s'agit d'une valeur de type div_t, c'est-à-dire une structure contenant deux champs entiers, quot et rem. Le signe du quotient est le produit des signes de p_1 et p_2, et le signe du reste est le même que celui de p_1. [voir aussi ldiv]

void________ exit(int) ________ <stdlib.h>

Cette fonction provoque la terminaison normale du programme avec transmission de la valeur p à l'environnement d'exécution. Les valeurs 0 et EXIT_SUCCES correspondent à une terminaison correcte, et EXIT_FAILURE à une terminaison sur erreur. La fonction exit est appelée automatiquement lors du retour de la fonction main. [voir aussi atexit]

double________ exp(double) ________ <math.h>

Cette fonction retourne la valeur de l'expression e^p.
Erreur : HUGEVAL *et* errno=ERANGE *ou* errno=EDOM

double________ fabs(double) ________ <math.h>

Cette fonction retourne $|p|$, la valeur absolue de p.
Erreur : HUGEVAL *et* errno=ERANGE *ou* errno=EDOM

int________ fclose(FILE*) ________ <stdio.h>

Cette fonction ferme le flot p; elle retourne EOF sur erreur et zéro sinon.
Erreur : EOF [voir aussi fopen,freopen]

int________ feof(FILE*) ________ <stdio.h>

Cette fonction retourne *vrai* si le flot p est positionné en fin de fichier.
[voir aussi clearerr]

int________ ferror(FILE*) ________ <stdio.h>

Cette fonction retourne *vrai* si une erreur s'est produite sur le flot p.
[voir aussi clearerr,perror,strerror]

int________ fflush(FILE*) ________ <stdio.h>

Cette fonction force l'écriture des données mémorisées dans le tampon du flot p.
Erreur : EOF [voir aussi fprintf,fputs,fputc]

int ______ **fgetc(FILE*)** ______ **<stdio.h>**

Cette fonction retourne un caractère lu sur le flot p, et EOF en cas d'erreur ou de détection de fin de fichier. Elle est l'équivalent, sous la forme d'une fonction véritable, de la macro getc.

Erreur : EOF [voir aussi getc,getchar,ungetc]

int ______ **fgetpos(FILE*, fpos_t*)** ______ **<stdio.h>**

Cette fonction écrit à l'adresse p_2 la position courante associée au flot p_1 et retourne une valeur non nulle sur erreur.

Erreur : $v \neq 0$ [voir aussi ftell,fsetpos,rewind]

char* ______ **fgets(char*, int, FILE*)** ______ **<stdio.h>**

Cette fonction lit une chaîne sur le flot p_3, jusqu'à un caractère \n, où jusqu'à ce que $p_2 - 1$ caractères ait été lus. Tous les caractères lus, y compris l'éventuel \n, sont placés à partir de l'adresse p_1, et un caractère \0 est rajouté comme dernier caractère. La fonction fgets retourne l'adresse p_1.

Erreur : NULL [voir aussi fopen,fscanf,fgetc]

double ______ **floor(double)** ______ **<math.h>**

Cette fonction retourne le réel de valeur $\lfloor p \rfloor$ (le plus grand entier inférieur ou égal à p).

Erreur : HUGEVAL *et* errno=ERANGE *ou* errno=EDOM [voir aussi ceil]

double ______ **fmod(double, double)** ______ **<math.h>**

Cette fonction retourne le plus petit, en valeur absolue, nombre réel r tel que $\frac{p_1-r}{p_2}$ soit un entier; le signe de r est celui de p_1, et p_2 doit être différent de zéro.

Erreur : HUGEVAL *et* errno=ERANGE *ou* errno=EDOM [voir aussi modf]

FILE* ______ **fopen(const char*, const char*)** ______ **<stdio.h>**

Cette fonction initialise un flot d'entrée-sortie, en ouvrant le fichier de nom p_1 dans le mode p_2 (r, w, a r+, w+ et a+). Le caractère b permet de sélectionner une entrée-sortie en mode binaire.

Erreur : NULL [voir aussi freopen,fflush,fclose]

int ______ **fprintf(FILE*, const char*, ...)** ______ **<stdio.h>**

Cette fonction formate la liste des paramètres éventuels à la suite de p_2, selon la spécification de format p_2, la chaîne de caractère résultante étant écrite sur le flot p_1. Elle retourne le nombre de caractères écrits, ou un nombre négatif sur erreur.

Erreur : $v < 0$ [voir aussi printf,sprintf,fscanf]

int ______ **fputc(int, FILE*)** ______ **<stdio.h>**

Cette fonction écrit sur le flot p_2 le caractère p_1 et retourne le caractère p_1, ou EOF en cas d'erreur. Elle est l'équivalent, sous la forme d'une fonction véritable, de la macro putc.

Erreur : EOF [voir aussi putc,putchar,fopen]

int ______ **fputs(char*, FILE*)** ______ **<stdio.h>**

Cette fonction copie la chaine p_1 sur le flot p_2 et retourne EOF si une erreur se produit.

Erreur : EOF [voir aussi puts,fprintf]

size_t ______ **fread(void*, size_t, size_t, FILE*)** ______ **<stdio.h>**

Cette fonction lit au plus p_3 objets de taille p_2 sur le flot p_4, et les copie à l'adresse p_1. Elle retourne le nombre d'objets effectivement lus.

Erreur : $v < p_3$ [voir aussi fwrite,ferror,feof]

void______ free(void*) ______ <stdlib.h>

Cette fonction libère la zone mémoire d'adresse p, sous condition qu'elle ait été précédemment allouée par malloc, calloc ou realloc et qu'elle n'ait pas été modifiée depuis lors; p peut valoir NULL. [voir aussi malloc,calloc,realloc]

FILE*______ freopen(const char*, const char*, FILE*) ______ <stdio.h>

Cette fonction modifie l'assignation du flot p_3 en l'associant à une entrée-sortie en mode p_2 sur le fichier de nom p_1.
Erreur : NULL [voir aussi fopen,fclose]

double______ frexp(double, int*) ______ <math.h>

Cette fonction calcule le réel $x \in [0.5, 1[$ et l'entier n tels que $p_1 = x \times 2^n$, et retourne la valeur x; la valeur n est recopiée à l'adresse p_2, et p_1 doit être différent de zéro.
Erreur : HUGEVAL *et* errno=ERANGE *ou* errno=EDOM [voir aussi ldexp,modf]

int______ fscanf(FILE*, const char*, ...) ______ <stdio.h>

Cette fonction analyse une suite de caractères lus sur le flot p_1, la découpe en tokens, et convertit chaque token en une valeur. Le découpage et la conversion sont effectués selon la spécification de format p_2. Pour chaque spécification de conversion que contient celle-ci, il doit exister un paramètre, passé par référence, dans la liste des paramètres éventuels à la suite de p_2. Chaque valeur résultant d'une conversion est copié à l'adresse du paramètre correspondant.
Erreur : EOF [voir aussi scanf,sscanf,fprintf]

int______ fseek(FILE*, long, int) ______ <stdio.h>

Cette fonction modifie la position courante associée au flot d'entrée-sortie p_1, selon le mode de positionnement spécifié par p_3 : SEEK_SET (à partir du début du fichier), SEEK_CUR (relativement à la position courante) et SEEK_END (à partir de la fin du fichier). Dans le cas d'une entrée-sortie en mode texte, la position p_2 est soit 0, soit une position précédemment obtenue par ftell utilisée en mode SEEK_SET. Dans le cas d'une entrée-sortie en mode binaire, p_2 est un nombre de caractères. Cette fonction ne peut être utilisée lorsque la taille du fichier est supérieure LONG_MAX. Dans ce cas, il est possible d'utiliser fgetpos et fsetpos.
Erreur : $v \neq 0$ [voir aussi ftell,fsetpos,rewind]

int______ fsetpos(FILE*, const fpos_t*) ______ <stdio.h>

Cette fonction affecte comme nouvelle position courante du flot p_1 la position p_2, précédemment obtenue par fgetpos.
Erreur : $v \neq 0$ [voir aussi fseek,fgetpos,rewind]

long______ ftell(FILE*) ______ <stdio.h>

Cette fonction retourne la position courante associée au flot d'entrée-sortie p. Elle échoue lorsque la taille à retourner est supérieure à LONG_MAX. Dans ce cas, il est possible d'utiliser la fonction fgetpos.
Erreur : -1L [voir aussi fseek,fgetpos,rewind]

size_t______ fwrite(const void*, size_t, size_t, FILE*) ______ <stdio.h>

Cette fonction écrit, sur le flot p_4, p_3 objets de taille p_2 rangés à partir de l'adresse p_1. Elle retourne le nombre d'objets effectivement écrits.
Erreur : $v < p_3$ [voir aussi fread,ferror,feof]

`int` —— `getchar(void)` —— `<stdio.h>`

Cette fonction est équivalente à `getc(stdin)`.
Erreur : `EOF` [voir aussi `getc,fgetc,ungetc`]

`int` —— `getc(FILE*)` —— `<stdio.h>`

(MACRO) Cette fonction retourne un caractère lu sur le flot p, et `EOF` en cas d'erreur ou de détection de fin de fichier. Elle est l'équivalent, sous la forme d'une macro, de la fonction `fgetc`.
Erreur : `EOF` [voir aussi `fgetc,getchar,ungetc`]

`char*` —— `getenv(const char*)` —— `<stdlib.h>`

Cette fonction retourne la valeur de la variable d'environnement de nom p.
Erreur : `NULL` [voir aussi `system`]

`char*` —— `gets(char*)` —— `<stdio.h>`

Cette fonction lit une ligne sur l'entrée standard, jusqu'à un caractère `\n` ou une fin de fichier, et la recopie à l'adresse p_1, en remplaçant le caractère `\n` par un caractère de fin de chaîne `\0`. Elle retourne p, ou `NULL` si une erreur s'est produite. Attention, si la chaîne lue contient un ou plusieurs caractères nuls, les caractères lus entre ce caractère et la fin de ligne seront perdus. D'autre part, il n'est pas possible de prévenir le débordement mémoire provoqué par la lecture d'une chaîne plus longue que la taille de la zone mémoire d'adresse p. Par conséquent, il vaut mieux ne pas utiliser cette fonction, qui n'a été maintenue que pour des raisons de compatibilité.
Erreur : `NULL` [voir aussi `fgets`]

`struct tm*` —— `gmtime(const time_t*)` —— `<time.h>`

Cette fonction retourne la conversion du temps interne p en un temps externe exprimé dans un système de mesure universel.
Erreur : `NULL` [voir aussi `time,localtime,mktime`]

`int` —— `isalnum(int)` —— `<ctype.h>`

Cette fonction retourne *vrai* si p appartient à l'alphabet `A-Z a-z 0-9` plus les éventuelles lettres locales et *faux* sinon. [voir aussi `tolower,toupper`]

`int` —— `isalpha(int)` —— `<ctype.h>`

Cette fonction retourne *vrai* si p appartient à l'alphabet `A-Z a-z` plus les éventuelles lettres locales, et *faux* sinon. [voir aussi `tolower,toupper`]

`int` —— `iscntrl(int)` —— `<ctype.h>`

Cette fonction retourne *vrai* si p est un caractère de contrôle (dans une implémentation ASCII, l'alphabet `\000-\037 \177`), et *faux* sinon.
[voir aussi `tolower,toupper`]

`int` —— `isdigit(int)` —— `<ctype.h>`

Cette fonction retourne *vrai* si p appartient à l'alphabet `0-9` et *faux* sinon.
[voir aussi `tolower,toupper`]

`int` —— `isgraph(int)` —— `<ctype.h>`

Cette fonction retourne *vrai* si p est un caractère affichable différent d'une espace, et *faux* sinon. [voir aussi `tolower,toupper`]

`int` —— `islower(int)` —— `<ctype.h>`

Cette fonction retourne *vrai* si p appartient à l'alphabet `a-z` plus les éventuelles minuscules locales, et *faux* sinon. [voir aussi `tolower,toupper`]

`int` ———— `isprint(int)` ———— `<ctype.h>`

Cette fonction retourne *vrai* si p est un caractère affichable, y compris une espace, et *faux* sinon. [voir aussi tolower,toupper]

`int` ———— `ispunct(int)` ———— `<ctype.h>`

Cette fonction retourne *vrai* si p est un caractère affichable différent d'une espace, d'une lettre ou d'un chiffre, et *faux* sinon. [voir aussi tolower,toupper]

`int` ———— `isspace(int)` ———— `<ctype.h>`

Cette fonction retourne *vrai* si p est l'un des caractères \f, \n, \r, \t, \v, ESPACE et *faux* sinon. [voir aussi tolower,toupper]

`int` ———— `isupper(int)` ———— `<ctype.h>`

Cette fonction retourne *vrai* si p appartient à l'alphabet A-Z plus les éventuelles majuscules locales, et *faux* sinon. [voir aussi tolower,toupper]

`int` ———— `isxdigit(int)` ———— `<ctype.h>`

Cette fonction retourne *vrai* si p appartient à l'alphabet 0-9 a-f A-F, et *faux* sinon. [voir aussi tolower,toupper]

`long` ———— `labs(long)` ———— `<stdlib.h>`

Cette fonction retourne $|p|$, la valeur absolue de p. [voir aussi abs]

`double` ———— `ldexp(double, int)` ———— `<math.h>`

Cette fonction retourne $p_1 \times 2^{p_2}$ (sur un argument entier, ce calcul est réalisé au moyen de l'opérateur <<).

Erreur : HUGEVAL *et* errno=ERANGE *ou* errno=EDOM [voir aussi frexp,modf]

`ldiv_t` ———— `ldiv(long, long)` ———— `<stdlib.h>`

Cette fonction retourne le résultat de la division entière $p_1 \div p_2$; il s'agit d'une valeur de type `ldiv_t`, c'est-à-dire une structure contenant deux champs entiers longs, `quot` et `rem`. Le signe du quotient est le produit des signes de p_1 et p_2, et le signe du reste est le même que celui de p_1. [voir aussi div]

`struct lconv*` — `localeconv(void)` ———— `<locale.h>`

Cette fonction retourne l'adresse d'une structure de type `struct lconv` contenant les informations de formatage de quantités numériques locales à l'implémentation : point décimal, séparateur de milliers, symbole monétaire, nombre de décimales dans une quantité monétaire, etc.

Erreur : NULL [voir aussi setlocale]

`struct tm*` —— `localtime(const time_t*)` ———— `<time.h>`

Cette fonction convertit la valeur de temps interne p dans un format appelé temps externe, décrit par la structure `struct tm`. Les champs de cette structure correspondent aux différentes informations exploitables de date et d'heure : année, mois, jour, jour du mois, heure, etc. [voir aussi time,asctime,mktime]

`double` ———— `log(double)` ———— `<math.h>`

Cette fonction retourne $\ln p$, le logarithme Neperien de p, la valeur de p doit être strictement positive.

Erreur : HUGEVAL *et* errno=ERANGE *ou* errno=EDOM

`double` ———— `log10(double)` ———— `<math.h>`

Cette fonction retourne $\log_{10} p = \frac{\ln p}{\ln 10}$, le logarithme décimal de p.; la valeur de p doit être strictement positive.

Erreur : HUGEVAL *et* errno=ERANGE *ou* errno=EDOM

`void` ——— `longjmp(jmp_buf, int)` ——— `<setjmp.h>`

Cette fonction restaure le contexte sauvegardé dans p_1 par un précédent appel à la fonction `setjmp`, en transmettant la valeur p_2. Cela provoque un saut assimilable à une cascade de `return` jusqu'à revenir au contexte d'appel à `setjmp`. Le dépilement de ce dernier contexte répète le retour de fonction de setjmp, avec cette fois la valeur p_2. [voir aussi `setjmp`]

`void*` ——— `malloc(size_t)` ——— `<stdlib.h>`

Cette fonction retourne l'adresse d'une zone mémoire allouée dynamiquement, et de taille p; elle retourne `NULL` si p égale à zéro, ou sur erreur.
Erreur : `NULL` [voir aussi `calloc,realloc,free`]

`int` ——— `mblen(const char*, size_t)` ——— `<stdlib.h>`

Cette fonction calcule et retourne la longueur en octets de la chaîne de multi-caractères pointée par p_1, en parcourant au plus p_2 caractères.
Erreur : `-1` [voir aussi `mbstowcs,mbtowc`]

`size_t` ——— `mbstowcs(const char*, wchar_t*, size_t)` ——— `<stdlib.h>` `<stddef.h>`

Cette fonction convertit la chaîne de multi-caractères commençant à l'adresse p_1 en une chaîne de caractères étendus, dont elle recopie au plus p_3 éléments à l'adresse p_2. Les zones source et destination ne doivent pas se recouvrir.
Erreur : `-1` [voir aussi `wcstombs,mblen`]

`int` ——— `mbtowc(wchar_t*, const char*, size_t)` ——— `<stdlib.h>` `<stddef.h>`

Cette fonction analyse le multi-caractère pointé par p_2, en parcourant au plus p_3 caractères, et calcule le code du caractère étendu correspondant. Si le code est trouvé, il est recopié à l'adresse p_1.
Erreur : `-1` [voir aussi `mblen,wctomb`]

`void*` ——— `memchr(const void*, int, size_t)` ——— `<string.h>`

Cette fonction parcourt les p_3 premiers caractères de p_1 et retourne l'adresse de la première occurrence du caractère p_2 rencontré si elle existe, et `NULL` sinon.
[voir aussi `strchr,strrchr`]

`int` ——— `memcmp((const void*, const void*, size_t))` ——— `<string.h>`

Cette fonction compare p_3 caractères à partir respectivement des adresses p_1 et p_2. À la différence de la fonction et `strncmp`, elle ne s'arrête pas lorsqu'elle rencontre un caractère `'\0'`. La valeur de retour à la même signification que celle de la fonction `strcmp`. [voir aussi `strcmp,strncmp`]

`void*` ——— `memcpy(void*, const void*, size_t)` ——— `<string.h>`

Cette fonction recopie à l'adresse p_1 les p_3 premiers caractères à partir de l'adresse p_2. Les chaînes de départ et d'arrivée sont supposées ne pas se recouvrir. À la différence de la fonction `strncpy`, elle ne s'arrête pas lorsqu'elle rencontre un caractère `'\0'`. [voir aussi `memmove,strcpy,strncpy`]

`void*` ——— `memmove(void*, const void*, size_t)` ——— `<string.h>`

Cette fonction à le comportement que `memcpy`, si ce n'est que les chaînes source et destination peuvent se recouvrir. [voir aussi `memcpy`]

`void*` ——— `memset(void*, int, size_t)` ——— `<string.h>`

Cette fonction recopie p_3 occurrences du caractère p_2 à partir de l'adresse p_1.
[voir aussi `memcpy`]

`time_t` ——— `mktime(struct tm)` ——— `<time.h>`

Cette fonction retourne la valeur de temps interne correspondant au temps externe p. C'est la fonction inverse de `time`.
Erreur : `-1` [voir aussi `time`,`localtime`]

`double` ——— `modf(double, double*)` ——— `<math.h>`

Cette fonction retourne la partie décimale de p_1 et recopie sa partie entière à l'adresse p_2; les deux valeurs sont du même signe que p_1.
Erreur : `HUGEVAL` *et* `errno=ERANGE` *ou* `errno=EDOM` [voir aussi `ldexp`,`fmod`]

`size_t` ——— `offsetof(struct` ⟨*s*⟩`,` ⟨*champs*⟩`)` ——— `<stddef.h>`

(MACRO)Cette fonction retourne la différence d'adresses en octets entre le début de la structure p_1 et le début du champs p_2 de cette structure (l'expression C $p_1.p_2$ doit être valide).

`void` ——— `perror(const char*)` ——— `<stdio.h>`

Cette fonction imprime la chaîne p, suivie du caractère :, et du message d'erreur associée à la valeur courante de `errno`. [voir aussi `strerror`]

`double` ——— `pow(double, double)` ——— `<math.h>`

Cette fonction retourne $p_1{}^{p_2}$ (les conditions initiales $\left\{\begin{array}{l} p_1 = 0 \\ p_2 \leq 0 \end{array}\right.$ et $\left\{\begin{array}{l} p_1 < 0 \\ p_2 \notin N \end{array}\right.$ sont exclues et provoquent une erreur).
Erreur : `HUGEVAL` *et* `errno=ERANGE` *ou* `errno=EDOM`

`int` ——— `printf(const char*, ...)` ——— `<stdio.h>`

Cette fonction est une abréviation pour `fprintf(stdout,` p`, ...)`.
Erreur : $v < 0$ [voir aussi `fprintf`,`sprintf`,`scanf`]

`int` ——— `putchar(int)` ——— `<stdio.h>`

Cette fonction est équivalente à `putc(`p`, stdout)`.
Erreur : `EOF` [voir aussi `putc`,`fputc`]

`int` ——— `putc(int, FILE*)` ——— `<stdio.h>`

(MACRO) Cette fonction écrit sur le flot p_2 le caractère p_1 et retourne le caractère p_1, ou `EOF` en cas d'erreur. Elle est l'équivalent, sous la forme d'une macro, de la fonction `fputc`.
Erreur : `EOF` [voir aussi `fputc`,`putchar`,`fopen`]

`int` ——— `puts(char*)` ——— `<stdio.h>`

Cette fonction recopie la chaîne p sur la sortie standard, en rajoutant à sa suite le caractère \n. Elle retourne `EOF` sur erreur. Elle n'est pas équivalente à `fputs(`p`, stdout)`, mais à `fputs(`p`, stdout), fputs("\n", stdout)`. Cette incompatibilité entre les deux fonctions a été conservée dans la norme ANSI pour préserver les programmes existant.
Erreur : `EOF` [voir aussi `fputs`,`printf`,`putc`]

`void` ——— `qsort(void*, size_t, size_t` ——— `<stdlib.h>`
`int (*)(const void*, const void *))`

Cette fonction trie le vecteur p_2 composé de p_3 éléments de taille p_4 en utilisant la relation d'ordre implémentée par la fonction p_4 (cette fonction devant respecter les conventions de `strcmp`); elle retourne le pointeur vers l'élément trouvé ou `NULL`. [voir aussi `bsearch`,`strcmp`]

int — **raise(int)** — **<signal.h>**

Cette fonction envoie le signal p au programme et retourne un valeur nulle si elle échoue.

Erreur : $v \neq 0$ [voir aussi signal,constantes SIG···]

int — **rand(void)** — **<stdlib.h>**

Cette fonction retourne un nombre pseudo-aléatoire compris entre 0 et RAND_MAX. Il n'y a pas de garantie d'obtenir toujours la même suite, pour toute implémentation. [voir aussi srand]

void* — **realloc(void*, size_t)** — **<stdlib.h>**

Cette fonction retourne un pointeur vers une zone mémoire de taille p_2, allouée dynamiquement, initialisée avec le contenu de la zone pointée par p_1 (si p_2 est inférieur à l'ancienne taille de p_1, seuls p_2 octets sont recopiés); elle retourne NULL et laisse $*p_1$ inchangé sur erreur ou si p_2 vaut 0.

Erreur : NULL [voir aussi calloc,malloc,free]

int — **remove(const char*)** — **<stdio.h>**

Cette fonction détruit le fichier de nom p.

Erreur : $v \neq 0$ [voir aussi rename]

int — **rename(const char*, const char*)** — **<stdio.h>**

Cette fonction renomme le fichier p_1 en lui donnant comme nouveau nom le nom p_2. Si le renommage implique une recopie, il est abandonné. Dans le cas où le fichier p_2 existe, le résultat dépend de l'implémentation.

Erreur : $v \neq 0$ [voir aussi remove]

void — **rewind(FILE*)** — **<stdio.h>**

Cette fonction réinitialise la position courante associée au flot p au début du fichier. [voir aussi fseek,fsetpos]

int — **scanf(const char*, ...)** — **<stdio.h>**

Cette fonction est une abréviation de fscanf(stdin, p,...).

Erreur : EOF [voir aussi fscanf]

int — **setjmp(jmp_buf)** — **<setjmp.h>**

Cette fonction sauvegarde le contexte d'exécution courant au moment de l'appel dans la variable p et retourne la valeur 0. [voir aussi longjmp]

char* — **setlocale(int, const char*)** — **<locale.h>**

Le mécanisme de définitions locales permet un paramétrage local de l'environnement en fonction des spécificités du pays. Ces paramètres sont classés en catégories : LC_CTYPE, LC_COLLATE, LC_TIME, LC_NUMERIC, LC_MONETARY, LC_MESSAGES. La fonction setlocale permet de modifier les mécanismes de la catégorie p_1 avec les informations décrites dans p_2. Si p_2 est le pointeur NULL, le mécanisme est désactivé.

Erreur : NULL [voir aussi localeconv]

void — **setbuf(FILE*, char*)** — **<stdio.h>**

Cette fonction affecte la zone mémoire d'adresse p_2 et de taille BUFSIZE au flot p_1. Si p_2 vaut NULL, la mémorisation est désactivée. [voir aussi fopen,setvbuf]

`int` ______ `setvbuf(FILE*, char*, int, size_t)` ______ `<stdio.h>`

Cette fonction permet de sélectionner le mode de mémorisation du flot p_1, selon la valeur de p_3 : pas de mémorisation (`_IONBF`), mémorisation ligne par ligne (`_IOLBF`), et mémorisation complète (`_IOFBF`). Le tampon de mémorisation est alloué par `setvbuf` si p_2 vaut `NULL`. Dans le cas contraire, la zone mémoire débutant à l'adresse p_2 et de longueur p_4 est affectée comme tampon.
Erreur : $v \neq 0$ [voir aussi `fopen`,`setbuf`]

`void (*)(int)` _ `signal(int, void (*)(int))` ______ `<signal.h>`

Cette fonction attache la fonction p_2 à l'événement *"réception du signal p_1"*. La fonction en question sera automatiquement appelée lorsque le signal est reçu. Elle retourne la valeur de la fonction précédemment attachée à ce signal. Un appel avec `SIG_IGN` (resp. `SIG_DFL`) entraîne un masquage (resp. une réinitialisation du traitement) du signal p_1. [voir aussi `raise`,constantes `SIG*`]

`double` ______ `sin(double)` ______ `<math.h>`

Cette fonction retourne $\sin p$, le sinus de l'angle p exprimé en radiant.
Erreur : `HUGEVAL` *et* `errno=ERANGE` *ou* `errno=EDOM`

`double` ______ `sinh(double)` ______ `<math.h>`

Cette fonction retourne $\sinh p$, le sinus hyperbolique de p.
Erreur : `HUGEVAL` *et* `errno=ERANGE` *ou* `errno=EDOM`

`int` ______ `sprintf(char*, const char*, ...)` ______ `<stdio.h>`

Cette fonction effectue, avec la spécification de format p_2, le même travail de formatage des paramètres variables que `fprintf`, à la différence qu'elle copie le résultat à l'adresse p_1. Elle retourne la taille de la chaîne construite.
Erreur : $v < 0$ [voir aussi `fprintf`,`sscanf`]

`double` ______ `sqrt(double)` ______ `<math.h>`

Cette fonction retourne $\sqrt{p}$, la valeur de p devant être positive ou nulle.
Erreur : `HUGEVAL` *et* `errno=ERANGE` *ou* `errno=EDOM`

`void` ______ `srand(unsigned int)` ______ `<stdlib.h>`

Cette fonction initialise avec la valeur p le générateur pseudo-aléatoire utilisé par la fonction `rand`. Par défaut, la suite est initialisée avec la valeur 1.
[voir aussi `rand`]

`int` ______ `sscanf(char*, const char*, ...)` ______ `<stdio.h>`

Cette fonction effectue, sur la chaîne p_1, et avec la spécification de format p_2, le même travail de décodage que `fscanf`.
Erreur : `EOF` [voir aussi `fscanf`]

`char*` ______ `strcat(char*, const char*)` ______ `<string.h>`

Cette fonction concatène la chaîne de caractères p_2 à la suite de la chaîne p_1 et retourne p_1; la zone mémoire pointée par p_1 doit être suffisamment grande et accessible en écriture. [voir aussi `strncat`]

`char*` ______ `strchr(const char*s, int)` ______ `<string.h>`

Cette fonction recherche la première occurrence du caractère p_2 dans la chaîne p_1; elle retourne l'adresse de ce caractère s'il existe, et le pointeur `NULL` sinon.
[voir aussi `strrchr`]

`int` ______ `strcmp(const char*, const char*)` ______ `<string.h>`
Cette fonction compare les chaînes de caractères p_1 et p_2 et retourne une valeur négative, nulle, ou positive, selon que la première est inférieure, égale, ou supérieure à la seconde, pour l'ordre lexicographique.
[voir aussi `memcmp`,`strncmp`]

`int` ______ `strcoll(const char*s, const char*s)` ______ `<string.h>`
Cette fonction compare les deux chaînes p_1 et p_2 en tenant compte de l'ordre lexicographique de l'implémentation locale.
Erreur : positionne `errno` [voir aussi `strxfrm`]

`char*` ______ `strcpy(char*, const char*)` ______ `<string.h>`
Cette fonction recopie la chaîne de caractères p_2 à l'adresse mémoire p_1 et retourne p_1; la zone mémoire pointée par p_1 doit être suffisamment grande et accessible en écriture. [voir aussi `memcpy`,`strncpy`]

`size_t` ______ `strcspn(const char*, const char*)` ______ `<string.h>`
Cette fonction retourne la longueur de la sous-chaîne préfixe maximale de la chaîne p_1 ne contenant aucun caractère de la chaîne p_2.
[voir aussi `strspn`,`strpbrk`,`strstr`]

`char*` ______ `strerror(int)` ______ `<string.h>`
Cette fonction retourne un pointeur vers le message d'erreur associé à l'erreur p.
[voir aussi `perror`]

`size_t` ______ `strftime(char*s, size_t,` ______ `<time.h>`
`const char*, const struct tm*)`
Cette fonction construit à l'adresse p_1 une chaîne de caractère de taille maximum p_2 résultat du formatage de la valeur p_4 exprimée en temps externe, selon la spécification de format p_3; retourne le nombre de caractères.
Erreur : 0 [voir aussi `time`,`localtime`,`asctime`]

`size_t` ______ `strlen(const char*s)` ______ `<string.h>`
Cette fonction retourne la longueur de la chaîne p, sans compter le caractère '\0' de fin de chaîne (si p pointe sur une chaîne vide `""`, `strlen` retourne 0). Par conséquent, la taille nécessaire pour coder une chaîne s est `strlen`$(s) + 1$.
[voir aussi `strcpy`,`mblen`]

`char*` ______ `strncat(char*, const char*, size_t)` ______ `<string.h>`
Cette fonction concatène au plus p_3 caractères de la chaîne p_2 à la suite de la chaîne p et retourne p_1; la zone mémoire pointée par p_1 doit être suffisamment grande et accessible en écriture (cette fonction peut recopier $p_3 + 1$ caractères).
[voir aussi `strcat`]

`int` ______ `strncmp(const char*,` ______ `<string.h>`
`const char*, size_t)`
Cette fonction compare les chaînes de caractères p_1 et p_2 sur au plus p_3 caractères, et retourne une valeur négative, nulle, ou positive, selon que la première est inférieure, égale, ou supérieure à la seconde, pour l'ordre lexicographique.
[voir aussi `memcmp`,`strcmp`]

`char*` ______ `strncpy(char*, const char*, size_t)` ______ `<string.h>`
Cette fonction recopie au plus p_3 caractères de la chaîne de caractères p_2 à l'adresse mémoire p_1 et retourne p_1, en complétant éventuellement avec des zéros si p_2 est trop courte; la zone mémoire pointée par p_1 doit être suffisamment grande et accessible en écriture. [voir aussi `memcpy`,`strcpy`]

`char*` ——— `strpbrk(const char*, const char*)` ——— `<string.h>`

Cette fonction recherche la première occurrence d'un des caractères de la chaîne p_2 dans la chaîne p_1 ; elle retourne son adresse si elle en trouve un, et `NULL` sinon.

[voir aussi strspn,strcspn,strstr]

`char*` ——— `strrchr(const char*s, int)` ——— `<string.h>`

Cette fonction recherche la dernière occurrence du caractère p_2 dans la chaîne p_1 ; elle retourne l'adresse de ce caractère s'il existe, et le pointeur `NULL` sinon.

[voir aussi strrchr]

`size_t` ——— `strspn(const char*, const char*)` ——— `<string.h>`

Cette fonction retourne la longueur de la sous-chaîne préfixe maximale de la chaîne p_1 ne contenant que des caractères de la chaîne p_2.

[voir aussi strcspn,strpbrk,strstr]

`char*` ——— `strstr(const char*, const char*)` ——— `<string.h>`

Cette fonction détermine si la chaîne p_2 est une sous-chaîne de la chaîne p_1 ; si c'est le cas, elle retourne l'adresse de sa première occurrence, et elle retourne `NULL` sinon. [voir aussi strspn,strcspn,strpbrk]

`double` ——— `strtod(const char*, char**)` ——— `<stdlib.h>`

Cette fonction retourne la valeur de la chaîne numérique réelle p_1, en défilant, s'il y a lieu, les *espaces* (voir fonction `isspace`) situés en tête. L'adresse de la fin de la chaîne est placée à l'adresses p_2 si elle est différente de `NULL`. La fonction `atof` est un cas particulier de `strtod` : `strtod(`⟨*s*⟩`, NULL)`.

Erreur : `errno = ERANGE` [voir aussi atof]

`char*` ——— `strtok(char*, const char*)` ——— `<string.h>`

Cette fonction recherche les mots de la chaîne p_1 en prenant comme ensemble de séparateurs les caractères de la chaîne p_2. Lors du premier appel, la fonction `strtok` modifie la chaîne p_1 en déplaçant le caractère `'\0'` à la fin du premier mot, et retourne l'adresse de ce mot. Les appels suivants doivent être effectués avec `NULL` comme premier paramètre, la fonction conservant la position courante dans la chaîne entre deux appels. Elle retourne `NULL` lorsqu'il n'y a plus de mot.

[voir aussi strchr,strstr]

`long` ——— `strtol(const char*, char**, int)` ——— `<stdlib.h>`

Cette fonction retourne la valeur de la chaîne numérique entière p_1 exprimée en base p_3, en défilant, s'il y a lieu, les *espaces* (voir fonction `isspace`) situés en tête. L'adresse de la fin de la chaîne est placée à l'adresses p_2 si elle est différente de `NULL`. Si p_2 vaut zéro, la base utilisée est déduite du format de la chaîne en utilisant les conventions C classiques : le préfixe `0x` pour la base 16, le préfixe `0` pour la base 8, et la base 10 sinon. Pour les bases supérieures à 10, les chiffres supplémentaires sont les caractères de `A` à `Z`. Par conséquent, la plus grande base exprimable est 36. La fonction `atol` est un cas particulier de `strtol` : `strtol(`⟨*s*⟩`, NULL, 10)`.

Erreur : `errno = ERANGE` [voir aussi atol,strtoul]

`unsigned long` ——— `strtoul(const char*, char**, int)` ——— `<stdlib.h>`

Cette fonction est équivalente à `strtol`, avec un résultat sans signe.

Erreur : `ULONGMAX` *et* `errno = ERANGE` [voir aussi atol,strtol]

`size_t` —— `strxfrm(const char*s, const char*s, size_t)` —— `<string.h>`

Cette fonction applique à la chaîne p_1 une transformation τ telle que la comparaison de deux chaînes quelconques s_1 et s_2 selon l'ordre lexicographique de l'implémentation locale, et la comparaison de $\tau(s_1)$ et $\tau(s_2)$ par la fonction `strcmp` donnent le même résultat. Les p_3 premiers caractères de $\tau(p_2)$ sont recopiés à l'adresse p_1. L'ordre lexicographique local tient compte des spécificités de l'alphabet en usage sur le site. Par exemple, sur un site français, les caractères `c` (code ASCII 99) et ç (code ASCII étendu 231) doivent être définis équivalent du point de vu de l'ordre lexicographique local.

Erreur : `-1` [voir aussi `strcoll`]

`int` —— `system(const char*)` —— `<stdlib.h>`

Cette fonction fait exécuter à l'interprète de commandes local la commande contenue dans la chaîne p. Le retour de la fonction `system` ne s'effectue que lorsque l'interprétation de cette commande est terminée, avec une valeur de retour dépendant de l'implémentation.

Erreur : `0 si non présent` (pour p=`NULL`)

`double` —— `tan(double)` —— `<math.h>`

Cette fonction retourne $\tan p$, la tangente de l'angle p exprimé en radiant.

Erreur : `HUGEVAL` *et* `errno=ERANGE` *ou* `errno=EDOM`

`double` —— `tanh(double)` —— `<math.h>`

Cette fonction retourne $\tanh p$, la tangente hyperbolique p.

Erreur : `HUGEVAL` *et* `errno=ERANGE` *ou* `errno=EDOM`

`time_t` —— `time(time_t*)` —— `<time.h>`

Cette fonction retourne la valeur du temps exprimé selon un codage interne au système, ou `-1` si cette fonctionnalité n'est pas disponible. Cette valeur est également recopiée à l'adresse p, sauf s'il s'agit du pointeur `NULL`. Le temps interne peut être converti en temps externe pour être exploité.

Erreur : `-1` [voir aussi `localtime,strftime,difftime`]

`FILE*` —— `tmpfile(void)` —— `<stdio.h>`

Cette fonction retourne un flot en mode lecture/écriture binaire sur un fichier vide. Le fichier est automatiquement détruit lors de sa fermeture.

Erreur : `NULL` [voir aussi `tmpname,fopen,fclose`]

`char*` —— `tmpnam(char [L_tmpnam])` —— `<stdio.h>`

Cette fonction produit un nom unique de fichier temporaire, de longueur maximum `L_tmpnam`, et le copie à l'adresse p, et la retourne. Elle génère au plus `TMP_MAX` fichiers différents.

Erreur : `NULL` [voir aussi `tmpfile,fopen`]

`int` —— `tolower(int)` —— `<ctype.h>`

Cette fonction retourne la majuscule du caractère p si p est une minuscule, et p sinon. [voir aussi `toupper,is···`]

`int` —— `toupper(int)` —— `<ctype.h>`

Cette fonction retourne la minuscule du caractère p si p est une majuscule, et p sinon. [voir aussi `toupper,is···`]

`int` ——— `ungetc(int, FILE*)` ——— `<stdio.h>`

Cette fonction replace le caractère p_1 dans le flot p_2 afin qu'il soit disponible pour une prochaine lecture. Un seul caractère par flot, différent de `EOF`, peut-être ainsi replacé.

Erreur : `EOF` [voir aussi `getc`,`fgetc`,`getchar`]

⟨*type*⟩ ——— `va_arg(va_list, ⟨type⟩)` ——— `<stdarg.h>`

La fonction `va_arg` retourne la valeur d'un paramètre de la liste des paramètres de la fonction courante F; p_2 est le nom du type de ce paramètre. L'appel est initialisé sur le premier paramètre à récupérer par `va_start`, et chaque appel retourne la valeur d'un nouveau paramètre, dans l'ordre dans lequel ils sont transmis à F. [voir aussi `va_start`,`va_end`]

`void` ——— `va_end(va_list)` ——— `<stdarg.h>`

La fonction `va_end` termine le parcours de paramètres d'une fonction en libérant d'éventuelles ressources allouées par `va_start`. [voir aussi `va_start`,`va_arg`]

`void` ——— `va_start(va_list, ⟨ident⟩)` ——— `<stdarg.h>`

La fonction `va_start` initialise la variable p_1 de telle façon que le prochain appel à la fonction `va_arg` avec p_1 en paramètre retourne la valeur du premier paramètre de la fonction courante qui suit le paramètre de nom p_2 dans la liste des paramètres de la fonction courante. [voir aussi `va_arg`,`va_end`]

`size_t` ——— `wcstombs(char*,` ——— `<stdlib.h>`
`const wchar_t*, size_t)` `<stddef.h>`

Cette fonction parcourt la chaîne de caractères étendus d'adresse p_2, construit la chaîne de multi-caractères correspondante, et en recopie au plus p_3 caractères à l'adresse p_1.

Erreur : `-1` [voir aussi `mbstowcs`,`mblen`,`wctomb`]

`int` ——— `wctomb(char*, wchar_t)` ——— `<stdlib.h>`
`<stddef.h>`

Cette fonction construit le multi-caractère, de taille maximale `MB_LEN_MAX`, codant le caractère étendu p_1, et le range à l'adresse p_2.

Erreur : `-1` [voir aussi `mbtowc`,`mblen`,`wcstombs`]

`int` ——— `vfprintf(FILE*, const char*, va_list)` ——— `<stdio.h>`
`<stdarg.h>`

Cette fonction est équivalente à `fprintf`, avec la différence que les paramètres à formater sont transmis sous la forme d'une liste `va_list`.

Erreur : $v < 0$ [voir aussi `printf`,`va_start`,`va_arg`]

`int` ——— `vprintf(const char*, va_list)` ——— `<stdio.h>`
`<stdarg.h>`

Cette fonction est équivalente à `printf`, avec la différence que les paramètres à formater sont transmis sous la forme d'une liste `va_list`.

Erreur : $v < 0$ [voir aussi `printf`,`va_start`,`va_arg`]

`int` ——— `vsprintf(char*, const char*, va_list)` ——— `<stdio.h>`
`<stdarg.h>`

Cette fonction est équivalente à `sprintf`, avec la différence que les paramètres à formater sont transmis sous la forme d'une liste `va_list`.

Erreur : $v < 0$ [voir aussi `printf`,`va_start`,`va_arg`]

Annexe B

Fichiers en-tête standard

assert.h
assert

locale.h
localeconv setlocale

ctype.h
isalnum iscntrl isgraph isprint isspace isxdigit toupper
isalpha isdigit islower ispunct isupper tolower

math.h
acos atan2 cosh fabs frexp log sinh tanh
acos atan cos floor ldexp modf sin tan
asin ceil exp fmod log10 pow sqrt

setjmp.h
longjmp
setjmp

signal.h
raise
signal

stdarg.h
va_arg va_start
va_end

stddef.h
offsetof

stdio.h
clearerr fgetpos fread fwrite putchar scanf tmpnam
fclose fgets freopen getchar putc setbuf ungetc
feof fopen fscanf getc puts setvbuf vfprintf
ferror fprintf fseek gets remove sprintf vprintf
fflush fputc fsetpos perror rename sscanf vsprintf
fgetc fputs ftell printf rewind tmpfile

stdlib.h
abort atoi div labs mbstowcs realloc strtoul
abs atol exit ldiv mbtowc srand system
atexit bsearch free malloc qsort strtod wcstombs
atof calloc getenv mblen rand strtol wctomb

string.h
memchr
memcmp memset strcmp strcspn strncat strpbrk strstr
memcpy strcat strcoll strerror strncmp strrchr strtok
memmove strchr strcpy strlen strncpy strspn strxfrm

time.h
asctime ctime gmtime localtime mktime strftime time
clock difftime

Annexe C

Table des codes ASCII

	dec.	hex.	oct.		dec.	hex.	oct.		dec.	hex.	oct.		dec.	hex.	oct.
C-@ (NUL)	0	00	000	␣	32	20	040	@	64	40	100	`	96	60	140
C-a (SOH)	1	01	001	!	33	21	041	A	65	41	101	a	97	61	141
C-b (STX)	2	02	002	"	34	22	042	B	66	42	102	b	98	62	142
C-c (ETX)	3	03	003	#	35	23	043	C	67	43	103	c	99	63	143
C-d (EOT)	4	04	004	$	36	24	044	D	68	44	104	d	100	64	144
C-e (ENQ)	5	05	005	%	37	25	045	E	69	45	105	e	101	65	145
C-f (ACK)	6	06	006	&	38	26	046	F	70	46	106	f	102	66	146
C-g (BEL)	7	07	007	'	39	27	047	G	71	47	107	g	103	67	147
C-h (BS)	8	08	010	(	40	28	050	H	72	48	110	h	104	68	150
C-i (HT)	9	09	011	)	41	29	051	I	73	49	111	i	105	69	151
C-j (LF)	10	0A	012	*	42	2A	052	J	74	4A	112	j	106	6A	152
C-k (VT)	11	0B	013	+	43	2B	053	K	75	4B	113	k	107	6B	153
C-l (FF)	12	0C	014	,	44	2C	054	L	76	4C	114	l	108	6C	154
C-m (CR)	13	0D	015	-	45	2D	055	M	77	4D	115	m	109	6D	155
C-n (SO)	14	0E	016	.	46	2E	056	N	78	4E	116	n	110	6E	156
C-o (SI)	15	0F	017	/	47	2F	057	O	79	4F	117	o	111	6F	157
C-p (DLE)	16	10	020	0	48	30	060	P	80	50	120	p	112	70	160
C-q (DC1)	17	11	021	1	49	31	061	Q	81	51	121	q	113	71	161
C-r (DC2)	18	12	022	2	50	32	062	R	82	52	122	r	114	72	162
C-s (DC3)	19	13	023	3	51	33	063	S	83	53	123	s	115	73	163
C-t (DC4)	20	14	024	4	52	34	064	T	84	54	124	t	116	74	164
C-u (NAK)	21	15	025	5	53	35	065	U	85	55	125	u	117	75	165
C-v (SYN)	22	16	026	6	54	36	066	V	86	56	126	v	118	76	166
C-w (ETB)	23	17	027	7	55	37	067	W	87	57	127	w	119	77	167
C-x (CAN)	24	18	030	8	56	38	070	X	88	58	130	x	120	78	170
C-y (EM)	25	19	031	9	57	39	071	Y	89	59	131	y	121	79	171
C-z (SUB)	26	1A	032	:	58	3A	072	Z	90	5A	132	z	122	7A	172
C-[(ESC)	27	1B	033	;	59	3B	073	[	91	5B	133	{	123	7B	173
C-\ (FS)	28	1C	034	<	60	3C	074	\	92	5C	134	|	124	7C	174
C-] (GS)	29	1D	035	=	61	3D	075	]	93	5D	135	}	125	7D	175
C-^ (RS)	30	1E	036	>	62	3E	076	^	94	5E	136	~	126	7E	176
C-_ (US)	31	1F	037	?	63	3F	077	_	95	5F	137	C-?	127	7F	177

Table des codes ASCII

Index des programmes

accind.c 401
addition.c 431
affect_vecteur.c 117
affiche_cars.c 33
affiche_cars.h 33
afficher.c 12
alarm_a.c 441
alarm_sv.c 442
allocation.c 228
append.c 348
arguments.c 37
argv_pp.c 185
assert.h 233
auto.c 201
autokill.c 434
autre_booleen.h 149
autre_erreur_decl.c 19
autre_triangle.c 272
av.2.c 419
av.c 416
avec_reg.c 68
avecpause.c 463

basename.c 219
booleen.c 149
booleen.h 149

c_anneau.c 294
c_anneau.h 292
calc.c 477
calcul.c 213
caractere_suivant.c 65
cellule.c 192
cellule.h 193
chaine_trier.c 215
chaines.c 182
chaines_concat.c 91
charger.c 429
classe.h 286
cles.c 189
Combien.c 24
compte_bits.c 109
compter.c 128
compteur.c 302
concatener.c 241
concatener.h 242
consult.c 404
cppif.c 168
creat.c 385

debug.h 165
defs.h 148
delta.c 26
deux_roues.c 290
dichotomie.c 139
divi.c 29
division.c 432
dmpchn.c 228
do.c 130
dynam_ini.c 81

e_suite.c 266
ecrire.c 340
effet_de_bord.c 120
encadre.c 33
err_aff_vecteur.c 93
err_contexte.c 252
erralloc.c 227
errdef.c 151
erreur_auto.c 186
erreur_decl.c 18
errno.c 326
es_controle.c 353
es_tests.c 347
execl.c 445
execle.c 449
execv.c 445
exemp_system.c 246
exit_if.h 157
exit_with.c 423
extropts.c 225

fnct_par.c 187
fork.c 451
fork_wait.c 461
format1.c 197
format2.c 198
format_etoile.c 200
formater.c 244
formater.h 244

gen_symb.c 262
gen_symb.h 261
generateur.c *(avec messages)* 266
generateur.c *(avec paramètres)* 268
generateur.c *(élémentaire)* 261, 263
generateur.c *(module fichier)* 271
generateur.h *(avec messages)* 265
generateur.h *(avec paramètres)* 267

generateur.h *(élémentaire)* 260
generateur.h *(module fichier)* 271
getchar.c 352
gros.c 420

Hal.c 25
handler.c 438
heure.c 218
hms.c 238

ident_fonc.c 94
ident_vect.c 92
il_est_environ.c 217
implementation_de_enr.c 110
inc_a.h 160
inc_ab.c 160
inc_b.h 160
ini_chn_vct.c 79
isdir.c 374
itob.c 229

kill0.c 436

lancer.c 446
lecture_controlee.c 367
line.c 169
LINEFILE.c 153
lire_chaine.c 222
lire_entier.c 365
lister.c 341
longuechaine.c 91
lsdates.c 371
lsdir.c 381
lsg.c 375
lstaille.c 373

main.c 12
main_divi.c 28
majuscule.c 212, 264
Makefile 13
malldmp.c 226
maxfd.c 337
marmot.c 136
memfaute.c 179
message.c 406
mode.c 377
motif.c 82
multiplication.c 431

nombre_d_elements.c 112
nomcmd.c 220
nomtemp.c 410
noredir.c 204
nultest.c 115

objet.h 284
optimisation.c 70
options.c 224
ouverture.c 338
ouvrir.c 334

p1.c 470
p1p2.c 469
p2.c 471
pere_fils.c 466
permanente.c 67
petit.c 420
pid.c 410
pile.c 319
pile.h 171, 319
pipecmd.c 475
plaque.c 304
pluriel.h 113
position.c 346
premier_programme.c 8
printexit.c 460
ptfonc.c 189
pyra.c 36

racine.c 235
racines.c 30
readwrite.c 342
recdef.c 159
recopie.c 341
redefinitions.c 149
redirection.c 350
rename.c 392
reopen.c 203
retenir.c 390
return_exit.c 142
roue_num.c 287
roue_num.c *(avec retenue)* 308
roue_num.h 285
roue_num.h *(avec gestion d'une retenue)* 307

saisie.c 403
sans_reg.c 68
sh_c.c 448
shell.c 452
sig2.c 440
signal.c 439
sp.c 394
spr.c 397
sq_genere.c 206
sq_lit.c 207
st.c 276
st.def 276, 278
stat.c 275
stat.c *(nouvelle implémentation)* 279
stat.h 275
subsep_decoupe.c 40
subsep_en_vrac.c 40
subsep_indent_1.c 41
subsep_indent_2.c 42
svaout.c 425
sw.c 126
sw_break.c 127

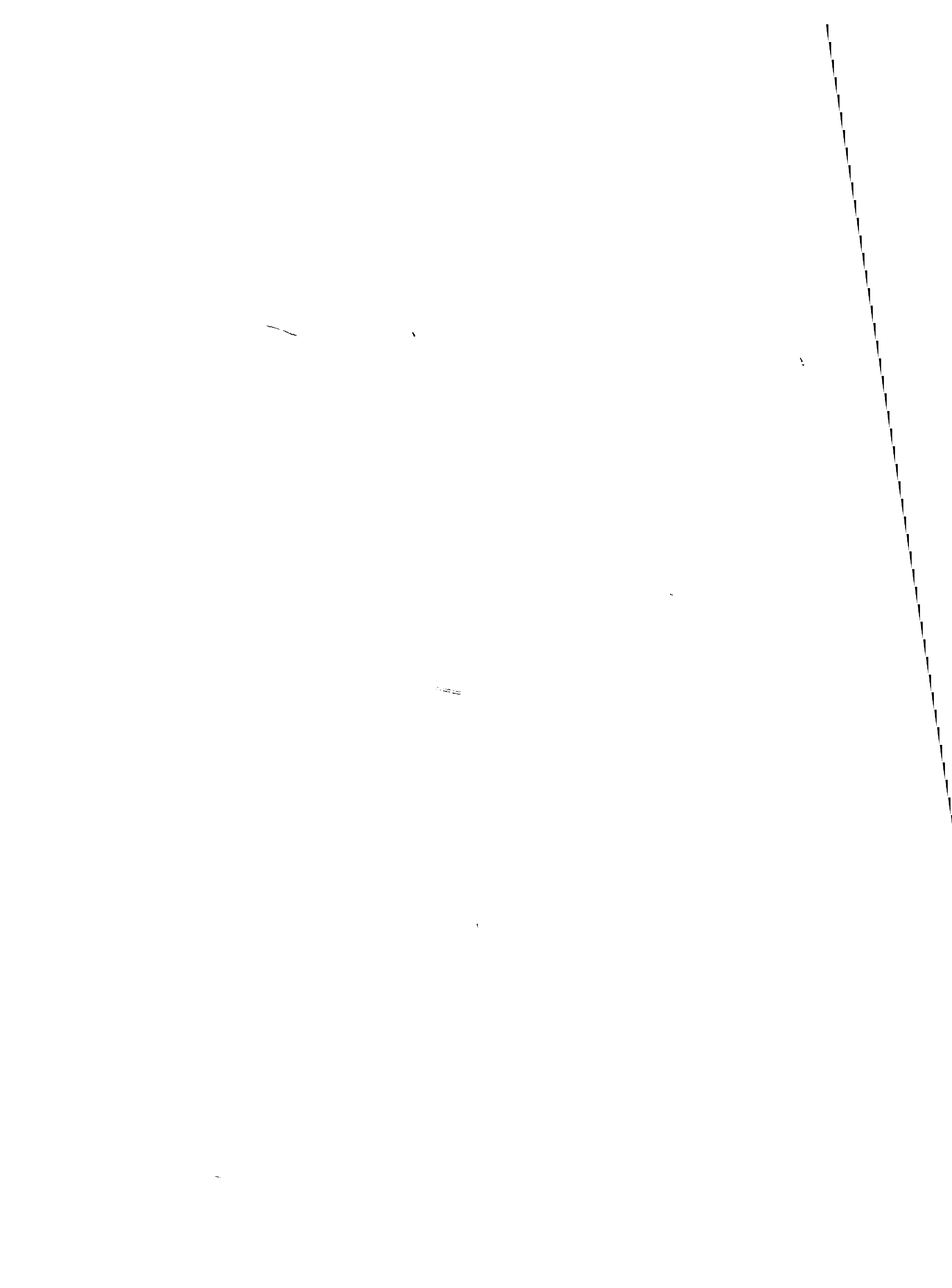
symboles.c 262
systeme.c 454

t_exit.c 158
taille.c 35
tampon.c 209
tchr.c 431
test_anneau.c 297
test_anneau.def 296
test_chaine_trier.c 216
test_concatener.c 242
test_formater.c 243
test_hms.c 238
test_instancier.c 287
test_lirechaine.c 223
test_pile.c 320
test_tube.c 344
test_typo.c 264
test_unites.c 151
test_usertime.c 414
test_vecteur.c 317
test_verrou.c 388
time_f.c 237
tliste.c 193
totalisateur.c *(avec liste de roues)* 300
totalisateur.c *(avec retenue)* 310
totalisateur.h 299
toucher.c 385
tpile.c 172
tpile.c_E 173
tracer.c 454
tread.c 339
tri3.c 74
tri3car.c 73
tri3int.c 74
triangle.c 133
triangle_croissant.c 267
triangle_decroissant.c 269
trigraph.c 89
trivial_es.c 355
tsv.c 427
tty.h 365
tty_bsd.h 360
tty_sv.h 362
tube.c 456
types.c 64
typo.c 264

ucreat.c 386
unites.h 150
unsigned.c 49
usertime.c 414

vecteur.c 316
vecteur.h 314
verrou.c 387
version.c 154
visibilite.c 75
volatile.c 72

while.c 129
whileplusplus.c 131



Index général

! 104
!= 104
" 90
23, 147, 158
158
% 103, 196
%= 117
& 27, 106, 109, 180
&& 104
&= 117
() 100, 101
* 103, 109
*/ 9
*= 117
+ 103
++ 114
+= 117
, 36, 113
- 103
-- 33, 114
-= 117
-> 100, 180
. 100, 180
... 23, 242
/ 103
/* 9
/= 117
: 59, 125
< 104
<< 107
<<= 117
<= 104
= 27, 115
== 104
> 104
>= 104
>> 107
>>= 117
?: 113
\ 91, 147
^ 106
^= 117
_ 86
} 122
| 106
|= 117
|| 104
{ 122
~ 106

a_bss 421
a_data 421
a_entry 422
a_syms 421
a_text 421
abort 229
abstraction
de constantes littérales 256, 276–278
modulaire 256
accès 379
access 383
adressage 325
adresse d'objets 51, 109
affectation 27, 103, 115
de pointeurs 176
de vecteurs 117
AFUU 11
aiguillage 125–129
alarm 441
allocation/rallongement 224, 228, 279
allocation dynamique 191, 221, 421
analyse
fonctionnelle descendante 256
par objets 256
appel de fonction 101
arbre de syntaxe 97
argc 24
arguments d'une commande 23, 37, 38
argv 24, 34, 184
arité 96
assembleur 10
assert 229
<assert.h> 162
assertions 229
associativité 98
atof 27, 213
atoi 213
atol 213
attributs 282
auto 67
auto-décrémentations 114
auto-incrémentations 114
avenrun 416

bibliothèque
mathématique 162
standard 195

- bloc 8, 122
- booléen 106
- boucles
  - pour 132
  - tant-que 129
- branchement inconditionnel 140
- break 126, 135
- brk 421
- BSS 420
- *buffer* 195

- C++ 20, 63
- C-d 351
- C-j 357
- C-m 357
- calloc 221
- caractères 48
  - étendus 52
  - multi-caractères 53
  - signés 50
- case 125
- CBREAK 360
- cc 10
- chaînage 299
- chaînes de caractères 90, 181
  - concaténation 214, 217
  - copie 214, 217
  - longueur 219
  - recherche d'un caractère 219
  - type 90
  - valeur 90
- champs de bits 59
- char 48
- chargement dynamique 429
- chdir 397, 413
- chemin 379
- chmod 372, 384, 411
- chown 384
- chroot 413
- classes 282
  - implémentation 286
  - interface 283, 284
- close 335
- closedir 381
- collisions de noms de fonction 285
- commentaires 9
- compatibilité ascendante 259, 269, 281, 307
- compilateur C 9
- constantes 85
  - caractère 88
  - entières 87
  - énumérations 50, 92
  - hexadécimales 87
  - littérales 85
    - abstraction 256, 276, 278
  - pointeur 175
  - réelles 88
  - symboliques 85, 92, 265
- contexte d'exécution 248
- continue 135
- conversion de type 100, 102, 103
- creat 332, 335, 385
- create 285
- CRMODE 360
- ctime 237
- <ctype.h> 162, 211

- __DATE__ 153
- d_fileno 379
- d_name 380
- d_namlen 379
- d_reclen 379
- dc 477
- débogueurs 166, 251, 422
- déclarations 47
  - définitions 76
  - références 76
  - visibilité 18
- default 125
- define 147
- defined 167
- delete 291
- descripteurs de fichier 331
- /dev/null 330
- *device driver* 329
- difftime 237
- directives 23, 147
- directory 381, 397
- DISPLAY 235
- do 129
- double 17, 50
- doubles définitions 75
- dup 473
- duplication de code 257

- ECHO 360
- effets de bord 26, 33, 113, 114, 118, 131, 260
- elif 164
- else 123
- emacs 86
- *emacs* 38
- en-tête 8, 23
- en-tête standard 161
- encapsulation de données 259
- endif 163
- entrées-sorties 163
  - accès direct 208, 399
  - de bas niveau 195
  - de haut niveau 195
  - détection de fin de fichier 345, 351, 466
  - duplication de descripteur 473
  - fermeture de fichier 202
  - fin de fichier 339, 351
  - flots 196
  - formatage 22, 23
  - immédiates 196

index de position 331
index de position courante 340, 342
initialisation 202
lectures non bloquantes 352
mémorisation des entrées 210
mémorisation des sorties 209
mémorisées 196
mode 331
mode canonique 361
mode *cbreak* 358
ouverture de fichiers 202
pilotes 329
redirections 349
standard 331
synchronisation 210, 355
tampon 353
tampons 209
validation 357
entrées-sorties généralisées 330
enum 50
énumération de constantes 50, 92
_environ 422
environnement 234, 449
envoi de message 265, 282, 285
envp 235, 449
EOF 35, 345, 351, 364
EOL 364
erase 356
erreurs
appels système 326
préprocesseur 155
sauvegarde de contexte 252
errno 326
<errno.h> 162
/etc 405
étiquette 140
euid 411
exclusion mutuelle 386
execl 444
execle 444, 449
execlp 444, 448
execv 444
execve 444, 449
execvp 444, 448
_exit 412
exit 29, 141, 422
expressions 16, 96
affectation 27, 103, 115–117, 176
arbre de syntaxe 97
auto-décrémentations 114
auto-incrémentations 114
constantes 77, 174
décalage 108
élémentaires 85
indirections 51, 111
listes d'expression 36
opérateurs 16
primaires 100
taille de 111
type et valeur 16, 97
extensibilité 255, 259
extern 273

F
suffixe réel simple précision 88
F_GETFL 353
F_OK 384
F_SETFL 353
factorisation de code 257
faux 106
fclose 202
fcntl 353
fflush 209, 356
fgetpos 209
fichiers 331
accès direct 346
accès indexé 399
attributs 369
chemin 379
core 180, 422, 433
création 385
descripteurs 413
droits d'accès 377, 383, 386
en-tête 161
en-tête 159
exécutables 10
fin de fichier 339, 351
liens 389
modification des attributs 384
objet 10, 11
source 9
spéciaux 329, 396
unité modulaire 273
file 376
__FILE__ 153
FILE 196
file descriptor 331
fin de fichier 339, 351, 466
fin de ligne 8
float 50
<float.h> 162
flot 196
fonctions 8, 53
bibliothèque 260
déclaration 17
d'instanciation 286
d'interface 273
globales 73
locales 73
main 8
mathématique 162
paramètre 120
paramètres 19
déclaration 20
de **main** 23
passage par référence 27
type 21

paramètres variables 162, 240
passées en paramètre 192, 298, 300
pointeur de 298
pures 259
type 138
valeur de retour 138
visibilité 73
fopen 202
for 36, 113, 132
fork 450
formatage 195
formatage de données 196
formats 258
spécification 196
fprintf 205
Free Software Foundation 11
freopen 203
fseek 208
fsetpos 209
ftell 208

g-valeurs 96, 114, 115
gcc 11–13, 20, 70, 163
options
-c compilation et assemblage seulement 11
-D définition de pseudo-constante 153
-E préprocesseur seulement 150
-g débogage 230
-I répertoire d'inclusion 160
-l spécification de bibliothèque 32
-M calcul de dépendances 15
-MM calcul de dépendances 15
-o nom du fichier résultat 10
-O optimisation du code généré 69
-S compilation seulement 70
-s suppression de la table de symboles 420
gdb 230
générateurs 260
généricité 170, 313
getchar 35
getegid 412
getenv 235
geteuid 412
getgid 412
getpid 410
getppid 410
getpwent 405
getuid 412
gid 328
GNU 10, 11, 20, 38, 155, 211, 230
GNU Emacs 41, 43–46, 86, 233, 425
goto 140
gtty 361

héritage 283
HOME 235

ICANON 364
identificateur
de groupe 328
d'utilisateur 328, 411
identificateurs 26, 85
collision 271
de fonction 94
de vecteur 92
if 123
if-else 30
ifdef 163
ifndef 164
implémentation standard 7
include 159
inclusion de fichier 159
indentation des programmes C 38, 41
index 221
indirections 51, 111
initialisations
variables permanentes 77
variables temporaires 81
vecteurs 78
inodes 369
instanciation 282, 283
instructions 8
d'échappement 134
élémentaires 121
int 48
interface 259, 273
interruption 162
ioctl 358
version BSD 359
version SYSTEM V 361
isalnum 211
isalpha 211
isascii 212
iscntrl 211
isdigit 133, 211
isgraph 211
islower 211
isprint 211
ispunct 211
isspace 211
isupper 211
isxdigit 211
itérations 33, 36, 129, 134, 306

jmp_buf 249

kill 356
kill 433–437, 461
killpg 434
kmem 330, 416

L
suffixe entier long 88
suffixe réel précision étendue 88
lex 170, 211
liens symboliques 331

limites 162
<limits.h> 162
__LINE__ 153
line 169
linefeed 357
link 389
linux 20
lisibilité 38, 86, 135, 255
listes 191, 299
  parcours de 306
ln 415
load average 416
<locale.h> 162
localeconv 162
localtime 236, 371
LOGNAME 235
long double 50
long int 48
longjmp 249
ls 370, 383
lseek 346, 399
lvalues 96

macro-assembleur 147
main 184, 423
maintenabilité 255, 259
make 13–15
Makefile 13, 15
malloc 191, 221
masquage de l'implémentation 259
<math.h> 31, 162
MAXNAMLEN 380
mémoire
  libération 291
  rallongement 224, 228, 279
  tampon 195
message
  envoi 265, 282, 285
  sélecteur 266
messages d'erreur 169
mesures 413
méthodes 282
mise au point 153
mkdir 397
mknod 396
mktime 237
modularité 76, 255
module 273
multi-caractères 53

new 285
newline 8
nlist 418
nm 94, 417
NOFILE 337
noms de fonction
  collisions 285
NULL 162

O_APPEND 333, 348
O_CREAT 333
O_EXCL 333
O_NDELAY 334
O_RDONLY 333
O_RDWR 333
O_TRUNC 333
O_WRONLY 333
objet courant 283
objets 282
  programmation par 256, 282
offsetof 110
open 332, 385
opendir 381
opérandes 96
opérateurs
  arité 96
  associativité 98
  postfixes 97
  préfixes 96
  priorité 98
  surcharge 100
  unaire 96
optimisation du code généré 69–73
ordre lexicographique local 53

P.P.O. 256, 282
<param.h> 416
paramètres de fonction 102, 120
  effectifs 101
  formels 20
parcours de liste 306
partage de code 282
passwd 405
PATH 235, 448
pause 438, 441, 463
perror 326
pgrp 411
pid 409
pile 170
  d'exécution 248, 252, 420
  générique 318
pipe 336, 456, 465
pipe 336
pointeurs 51, 53, 55–58, 101
  addition 178
  affectation 176
  comparaison 177
  conversion 176
  de fonctions 57, 80, 298
  de vecteurs 56
  génériques 176, 314
  indirections 51, 111
  initialisation 179
  valeur 175
polymorphisme 283, 321, 322
portabilité 255
ppid 410
préprocesseur 23, 147

définition de pseudo-constantes 147–153
définition de pseudo-fonctions 154, 159
en-tête standard 161
erreurs classiques 155
généricité 171
inclusion de fichiers 159
pseudo-expressions 167
récursivité des définitions 159
prétraitement 147
printenv 235
printf 7, 21, 205
priorité des opérateurs 98
processus 409
chargement 444
communication par tube 465–472
contexte 454
duplication 450
groupe 411
identification 411
lancement 444
point d'entrée 422
région de la pile 248
région des données 248, 421
régions 420
région texte 221, 420
signaux 412
synchronisation 452, 462
terminaison 141, 412, 422, 459
terminaison anormale 433
programmation par objets 256, 282
programmation visuelle 312
programme exécutable 415
prompt 10
prototype 21, 161, 261, 273
ps 410
pseudo-constante 147
pseudo-fonctions 113, 147
ptrdiff_t 52, 162

quotation 201

R_OK 384
raise 162
ramasse-miettes 291
rand 16
RAW 360
read 338
readdir 381
receveur 282, 283, 286
recv 286
redirections 338, 454
dans un tube 472
référence 18
register 68
rename 393
répertoire 378–384
courant 413
entrée 379
listage 378–381
suppression 397
return 17, 135, 138
réutilisabilité 255, 259, 298
rindex 221
rm 14
rmdir 397
root 328, 412

S_IFBLK 374
S_IFCHR 374
S_IFDIR 374
S_IFIFO 374, 396
S_IFLNK 374
S_IFMT 374
S_IFREG 374
S_IFSOCK 374
sauvegarde
de contexte 249
d'image mémoire 424
sbrk 421
scanf 27
seekdir 381
segments de programme 415
sélecteur de message 266
setjmp 249
<setjmp.h> 162
setlocale 162
set uid bit 411
shell 448, 453
SHELL 235
short int 48
SIG_DFL 438
SIG_IGN 438
SIGALARM 441
SIGINT 439
SIGKILL 468
signal 162, 437
<signal.h> 162
signaux 162, 412, 433
alarme 441
écriture dans un tube 471
envoie 434
récupération 438
signed 50
SIGPIPE 466
SIGQUIT 439
SIGUSR1 463
size_t 52, 162
sizeof 111
sleep 353, 463
sprintf 201
sqrt 31
st_atime 371
st_ctime 371
st_dev 371
st_gid 371
st_ino 370
st_mode 370, 374

st_mtime 371
st_nlink 371
st_size 371
st_uid 371
Stallman Richard 11, 38
start 422
stat 370
static 29, 66, 73, 76, 271
<stdarg.h> 162, 240
__STDC__ 153, 165
<stddef.h> 162
stderr 196
stdin 196
stdout 196
strcat 214, 217
strchr 219
strcmp 214
strcpy 214, 217
stream 196
strftime 237
string 181, 214
<string.h> 214
strip 420
strlen 34, 219
strncat 215
strncmp 215
strncpy 215
strrchr 219
strtod 214
strtol 214
struct 58
struct exec 416
struct sgttyb 359
struct stat 370
struct termio 361
struct tm 236
struct tms 414
structures 54, 58
  champs 55
structures de contrôle 123
  for 36, 132
  tests 123
stty 356, 358, 361
super-utilisateur 328
surcharge d'opérateurs 100
swapper 409
switch 125
sys_errlist 326
system 246

table de symboles 415
temps 236
  interne 236
  externe 236
TERM 235
termcap 405
terminaison anormale 422, 435
termio 361
tests 30, 123
__TIME__ 153
time 69, 443
times 413
TIOCGETP 360
TIOCSETP 360
tms_cstime 413
tms_cutime 413
tms_stime 413
tms_utime 413
toascii 212
tolower 212
touch 15, 372, 384
toupper 212
transparence référentielle 260
trigraphs 88
tubes 336, 344, 455, 465
  nommés 396
  redirection 472
types 162
  caractère 48
  caractère étendu 52
  de base 47, 102
  définition 62
  d'un caractère 211
  d'une expression 97
  élémentaires 50
  entiers 48
  expressions de type 64, 65
typedef 64, 284

U
  suffixe entier sans signe 88
uid 328, 411
umask 386, 413
undef 148
union 60
UNIX 20
  appels système 325
  noyau 325
unlink 388, 389
unsigned 48
usage 142, 220
USER 235
utimes 384
utmp 405

va_arg 240
va_end 240
va_list 240
va_start 240
<varargs.h> 240
variables 26, 95
  automatiques 67
  d'instance 282
  durée de vie 66
  globales 28, 76, 273
  implémentation 248
  initialisation 26, 66, 77, 81
  locales 76, 261, 271

niveau 28, 65
permanentes (statiques) 66
registres 68
temporaires(dynamiques) 66
vecteurs 24, 53–58, 92, 101
affectation 117
extensibles 314
initialisation 78
verrouillage de ressource 386
visibilité 47, 65, 73–76
des déclaration 18
des fonctions 73
des variables 75
void 17
void * 314
volatile 72
vrai 106

W_OK 384
wait 452, 459, 461
wc 128
wchar_t 52, 162
while 33, 129
write 338

X_OK 384

yacc 170, 211

MASSON éditeur
120, Bd St-Germain
75280 Paris Cedex 06
Dépôt légal : février 1995

Normandie Roto Impression s.a.
61250 Lonrai
N° d'imprimeur : I5-0184
Dépôt légal : février 1995

SYSTÈMES D'EXPLOITATION

UNIX. Programmation avancée. M.J. Rochkind.
UNIX système V. Système et environnement. A.-B. Fontaine et Ph. Hammes.
LES SYSTÈMES D'EXPLOITATION. Structure et concepts fondamentaux. C. Lhermitte.
LE SYSTÈME D'EXPLOITATION PICK. M. Bull.

MICROPROCESSEURS ET ARCHITECTURE DE L'ORDINATEUR

TECHNOLOGIE DES ORDINATEURS pour les IUT et BTS. Avec exercices. P.-A. Goupille.
ARCHITECTURE DES ORDINATEURS. Des techniques de base aux techniques avancées. G. Blanchet et B. Dupouy.
PRINCIPES DE FONCTIONNEMENT DES ORDINATEURS. R. Dowsing et F. Woodhams.
IMPLANTATION DES FONCTIONS USUELLES EN 68000. F. Bret.
LES MICROPROCESSEURS 80286/80386. Nouvelles architectures PC. A.-B. Fontaine et F. Barrand.

RÉSEAUX ET TÉLÉCOMMUNICATIONS

L'UNIVERS DES RÉSEAUX ETHERNET. Concepts, produits, mise en pratique. N. Turin.
TÉLÉMATIQUE. Téléinformatique et réseaux. M. Maiman.
ARCHITECTURES DE COMMUNICATIONS (2 tomes). M. Maiman.
RNIS. Description technique. Ph. Chailley et D. Seret.
UNE INTRODUCTION À TCP/IP. J. Davidson.
X 25. Protocoles pour les réseaux à commutation de paquets. R.J. Deasington.
OSI. LES NORMES DE COMMUNICATION ENTRE SYSTÈMES OUVERTS. J. Henshall et S. Shaw.
LES RÉSEAUX LOCAUX. Comparaison et choix. J.S. Fritz, C.F. Kaldenbach, L.M. Progar.
ORDINATEURS INTERFACES ET RÉSEAUX DE COMMUNICATION. S. Collin.
RNIS. Concept et développement J. Ronayne.
LE RÉSEAU SNA. K.C.E. Gee.
RÉSEAUX ET MICRO-ORDINATEURS. Ph. Jesty.

INTELLIGENCE ARTIFICIELLE

INTELLIGENCE ARTIFICIELLE. E. Rich.
PROGRAMMATION DES SYSTÈMES EXPERTS EN PASCAL. B. Sawyer et D. Foster.
SYSTÈMES EXPERTS. Concepts et exemples. J.L. Alty et M.J. Coombs.

INFORMATIQUE POUR L'ENTREPRISE

COMPRENDRE, CONCEVOIR ET UTILISER LES SIAD. A. Checroun.
INFORMATIQUE DOCUMENTAIRE. A. Deweze.
INGÉNIERIE DES DONNÉES. Bases de données, systèmes d'information, modèles et langages. E. Pichat et R. Bodin.
MODÉLISATION DANS LA CONCEPTION DES SYSTÈMES D'INFORMATION. Avec exercices commentés. Acsiome.
L'AUDIT INFORMATIQUE. Méthodes, règles, normes. M. Thorin.
APPLICATION SYSTEM. Un système IBM de 4e génération. J. Rambaud.
COMPRENDRE LES BASES DE DONNÉES. Théorie et pratique. A. Mesguich et B. Normier.

INFORMATIQUE INDUSTRIELLE ET APPLICATIONS SCIENTIFIQUES

GRAPHISME DANS LE PLAN ET DANS L'ESPACE AVEC TURBO PASCAL 4.0. R. Dony.
SPÉCIFICATION ET CONCEPTION DES SYSTÈMES. Une méthodologie. J.-P. Calvez.
SPÉCIFICATION ET CONCEPTION DES SYSTÈMES. Études de cas. J.-P. Calvez.
PROGRAMMATION EN INFOGRAPHIE. Principes, exercices et programmes en C. L. Ammeraal.
INFOGRAPHIE ET APPLICATIONS. T. Liebling et H. Röthlisberger.
MÉTHODES DE DÉVELOPPEMENT D'UN SYSTÈME À MICROPROCESSEURS. A. Amghar.
INFORMATIQUE GRAPHIQUE DANS LE BATIMENT ET L'ARCHITECTURE. G. Lauret.
EXERCICES DE RECONNAISSANCE DE FORMES PAR MICRO-ORDINATEUR. Ph. Fabre.
SYSTÈMES TEMPS RÉEL EN ADA. Une approche virtuelle et asynchrone. L. Briand.
BASES DE DONNÉES POUR LE GÉNIE LOGICIEL. C. Godart et F. Charoy.
NORMES DE MÉTAFICHIERS ET D'INTERFACES POUR L'INFOGRAPHIE. D.B. Arnold et P.R. Bono.

MANUELS INFORMATIQUES MASSON

Méthodologie de la programmation en langage C

Principes et applications

J.-P. BRAQUELAIRE

Outre une définition syntaxique du langage, ce cours de programmation en langage C norme ANSI présente les principes, techniques et méthodes qui ont peu à peu vu le jour au sein de la communauté des utilisateurs du langage C.
La première partie de l'exposé est consacrée aux concepts de base sur lesquels repose ce langage : les fonctions, les constructions d'objets et de types, les expressions et les instructions. La seconde partie traite de l'environnement de programmation, c'est-à-dire le pré-processeur et la bibliothèque standard, et l'utilisation des pointeurs. Enfin, la dernière partie décrit la mise en œuvre des appels système Unix BSD et System V : entrées-sorties, système de fichiers, processus et gestion de la mémoire. Certaines de ces fonctionnalités, notamment les entrées-sorties, existent sous d'autres systèmes que le système UNIX (MS-DOS, VMS, GEM, MAC-OS...).
Le propos est illustré par des exemples nombreux et réalistes, présentés sous forme d'extraits de sessions de travail sous UNIX. Par ailleurs, l'auteur décrit l'environnement de programmation du domaine public et les logiciels du projet GNU, qui jouissent d'un succès de plus en plus important au sein de la communauté des programmateurs C et UNIX.
En outre, cette deuxième édition met l'accent sur la modularisation des programmes C et leur orientation objet. La norme ANSI étant actuellement effective, l'ouvrage comporte la bibliothèque standard C complète.
Ce cours est principalement destiné aux étudiants de second et troisième cycles d'informatique, aux élèves d'écoles d'ingénieurs et aux professionnels de l'informatique désirant s'initier aux principes de base de la programmation en C, ou approfondir leurs connaissances en la matière.

Jean-Pierre ***BRAQUELAIRE*** *enseigne l'informatique à l'université de Bordeaux I. Spécialiste du système UNIX et du langage C, ses recherches portent actuellement sur l'informatique graphique.*