



Outils de développement logiciel





Informatique, 1re année

Régis Clouard & Alain Lebret

2024

Table des matières

Avant-propos	1
1 Introduction	3
1.1 Informatique et métier de l'ingénieur	3
1.2 Objectifs du cours	4
1.3 ODL dans la formation	4
1.4 Contenu et modalités	5
2 Ordinateurs et systèmes d'exploitation	7
2.1 Introduction	7
2.2 Architecture matérielle et logicielle d'un ordinateur	8
2.3 Autour de l'unité centrale	9
2.4 Environnements	9
2.5 Logiciels	11
2.6 Accès aux informations	12
3 Connexion	15
3.1 Compte personnel	15
3.2 Connexion : mot de passe	16
3.3 Ouverture d'une session Ubuntu	18
4 Organisation des disques	19
4.1 Structure arborescente des disques	19
4.2 Dossier ou répertoire	19
4.3 Fichier	20
4.4 Protection des dossiers et fichiers	21
5 Commandes Unix	23
5.1 Introduction	23
5.2 Documentation	25
5.3 Gestion des dossiers et des fichiers	25
5.4 Gestion des processus	31
5.5 Archivage et compression	32
5.6 Réseau	33
5.7 Commandes diverses	35
5.8 Complément sur la protection des fichiers	35
5.9 tmux : un multiplexeur de terminal	36
 Exercices	38
6 Vim : un éditeur évolué pour développeur	41
6.1 Introduction	41
6.2 Intérêt pour le développement logiciel	42
6.3 Lancement de l'éditeur	42

6.4	Modes principaux de Vim	42
6.5	Commandes de Vim	43
6.6	Fenêtre principale de Vim	43
6.7	Mode insertion	44
6.8	Déplacement du curseur	44
6.9	Opérations standards	45
6.10	Opérations avancées	47
6.11	Configuration de Vim	50
6.12	Conclusion	51
	Exercices	52
7	Interpréteur de commandes	55
7.1	Introduction à Unix et aux interpréteurs de commandes	55
7.2	Différentes familles d'interpréteurs de commande	55
7.3	Fichiers de configuration	56
7.4	Gestion des commandes et historique	57
7.5	Exécution des commandes	58
7.6	Gestion des entrées-sorties et redirections	59
7.7	Variables du <i>shell</i>	60
7.8	Fichiers de scripts <i>shell</i>	62
7.9	Commande test	62
7.10	Structures de contrôle	64
7.11	Fonctions	67
7.12	Métacaractères du <i>shell</i>	68
7.13	Commandes spécifiques du <i>shell</i>	68
7.14	Calculs en bash	69
7.15	Conclusion	70
	Exercices	70
8	Édition de programmes	71
8.1	Introduction	71
8.2	Écriture d'un programme	72
8.3	Formatage du code dans Vim	80
8.4	Génération de la documentation avec Doxygen	81
	Exercices	82
9	Compilation	85
9.1	Étapes de compilation	85
9.2	Traduction	86
9.3	Optimisation	87
9.4	Assemblage	89
9.5	Édition des liens	90
9.6	Précompilation	91
9.7	Compilation multifichiers	94
9.8	Bibliothèques	96
9.9	Vim et la compilation	98
	Exercices	98
10	Commande make	101
10.1	Objectif	101




10.2	Structure d'un fichier Makefile	102
10.3	Commande make	103
10.4	Cibles	103
10.5	Dépendance	104
10.6	Commande dans une règle	104
10.7	Règles explicites	105
10.8	Macros	106
10.9	Règles implicites	107
10.10	Directives	109
10.11	Options de la commande make	110
10.12	Makefile et Vim	111
10.13	Outils de création de makefile	111
10.14	Exemple complet	113
	Exercices	114
11	Mise au point du logiciel	117
11.1	Introduction	117
11.2	Détection des anomalies	117
11.3	Détection des erreurs d'exécution	118
11.4	Détection des fuites de mémoire	120
11.5	Détection des erreurs d'algorithme	122
11.6	Tests	124
11.7	Optimisation du code	126
11.8	Notion de qualimétrie logicielle	127
	Exercices	128
12	Produit final	133
12.1	Entrées-sorties d'un programme	133
12.2	Organisation d'une distribution logicielle	134
12.3	Construction de l'archive de la distribution	135
12.4	Produit final	135
13	Gestion de version avec Git	137
13.1	Introduction	137
13.2	Configuration de Git	137
13.3	Qu'est-ce qu'un dépôt Git?	137
13.4	Commandes de base	138
13.5	Branches dans Git	139
13.6	Travail avec un dépôt distant	139
13.7	Gestion des conflits	140
13.8	Exemple complet	140
13.9	Conclusion	141
	Exercices	142
14	Éléments de réponse aux exercices	145
	Index	155
	Bibliographie	159

Table des figures

1.1	Logique d'organisation des cours.	4
2.1	Une machine et son programme intégré.	7
2.2	Un ordinateur.	7
2.3	Marché des systèmes d'exploitation 2009-2024 (source : gs.statcounter.com).	10
4.1	Arborescences sur Unix et MsWindows.	19
5.1	Représentation d'une commande Unix avec ses entrées-sorties par défaut.	24
5.2	Exemple d'un pipeline de commandes.	24
5.3	Structure de tmux.	36
5.4	Exemple de composition avec tmux.	37
6.1	Environnements populaires chez les développeurs en 2021 (source : stackoverflow.com).	41
6.2	Modes principaux de Vim.	43
6.3	Fenêtre de l'éditeur Vim en mode insertion.	44
7.1	Architecture en couches d'Unix.	55
9.1	Chaîne de compilation permettant de produire l'exécutable « bonjour ».	86
9.2	Chaîne de compilation : étape de traduction.	87
9.3	Chaîne de compilation : étape d'optimisation.	87
9.4	Chaîne de compilation : étape d'assemblage.	90
9.5	Chaîne de compilation : étape d'édition de liens.	90
9.6	Chaîne de compilation : étape de précompilation.	91
9.7	Bibliothèques statiques et dynamiques.	97
10.1	Exemple de logiciel composé de plusieurs fichiers avec leurs dépendances.	101
10.2	Diagramme de dépendances du jeu worms.	115
11.1	Vim en mode déverminage avec GNU gdb.	120
11.2	Parcours en <i>zigzag</i> de la matrice.	129

Avant-propos

Dans ce document, vous trouverez trois types de paragraphes particuliers permettant de mettre en évidence des points importants, des remarques et des astuces.



Un point important.



Une remarque.



Une astuce.

D'autre part, des exemples de code qui illustrent le cours sont disponibles à l'adresse suivante :

<https://github.com/alainlebreton/odl>

1 Introduction

« Measuring programming progress by lines of code is like measuring aircraft building progress by weight.
» > Bill Gates.

1.1 Informatique et métier de l'ingénieur

L'informatique (*computer science* en anglais), est la science du traitement de l'information. Un logiciel (*a software*), quant à lui, est un programme ou un ensemble de programmes exécutables par un ordinateur et qui permet de traiter automatiquement une tâche définie par un client. Le logiciel est ainsi le coeur du métier d'un ingénieur en informatique, car il constitue la solution concrète aux besoins exprimés par le client.

Métier

Le métier d'ingénieur en informatique englobe plusieurs domaines clés. Tout d'abord, la gestion de projet est essentielle, incluant la gestion du temps, des coûts et de l'organisation. Ensuite, la modélisation de solutions permet de conceptualiser des réponses aux problématiques posées. Le développement de la solution, souvent sous forme de programme, est une étape cruciale, surtout au début de la carrière. Enfin, la qualité et la maintenance des logiciels assurent leur pérennité et leur efficacité à long terme.

Compétences requises

Les compétences requises pour un ingénieur en informatique sont variées et incluent à la fois des qualités humaines et des compétences techniques. Sur le plan humain, la gestion d'équipe est primordiale. Il est important de dépasser sa timidité et de savoir chercher l'information. Il est en effet formellement interdit pour un ingénieur de rester bloqué trop longtemps sur un problème sans solliciter l'aide de ses collègues, car la collaboration est essentielle. Les capacités d'analyse sont également indispensables pour comprendre et résoudre les problèmes complexes. Quant aux compétences techniques, elles sont nécessaires non seulement pour la crédibilité de l'ingénieur, mais aussi pour la qualité de ses interactions avec ses collègues.

Domaine d'intervention

L'ingénierie informatique touche à tous les secteurs. Les métiers accessibles aux ingénieurs en informatique sont divers et passionnants. Par exemple, dans le domaine de l'image et du son, un ingénieur peut travailler sur la réalité virtuelle, les jeux vidéo, et l'interaction homme/machine. Dans le secteur bancaire, il peut se spécialiser dans les systèmes de paiement électronique. En cybersécurité, l'ingénieur contribue à la protection des données et des systèmes. Enfin, en intelligence artificielle, il développe des solutions innovantes pour des applications variées, allant de la reconnaissance vocale à la conduite autonome.

1.2 Objectifs du cours

Le cours d'outils de développement logiciel a pour objectif d'amener à la maîtrise des environnements et outils de développement de logiciels. Plus précisément, il vise à :

1. Initier les étudiants, depuis la découverte du clavier jusqu'à la programmation de logiciels complets.
2. Unifier les compétences des étudiants provenant de diverses formations. Pour ceux issus des classes préparatoires, il s'agit d'apprendre les bases de la construction de logiciels. Pour les étudiants d'IUT et de licences, le cours vise à approfondir et valider leurs compétences, facilitant ainsi leur transition du statut de technicien à celui d'ingénieur.
3. Développer l'autonomie des étudiants sur un système informatique. Ils doivent être capables de configurer leur environnement, d'installer et de configurer des logiciels et des matériels, et de résoudre les problèmes liés à l'utilisation du système, comme la récupération de fichiers endommagés ou la réalisation de sauvegardes.
4. Connaître les capacités du système d'exploitation de la machine sur laquelle ils travaillent.
5. Améliorer les performances de développement logiciel en termes de robustesse, rapidité et portabilité, permettant ainsi de créer des logiciels multiplateformes efficaces.



Le cours progresse rapidement en complexité, ce qui nécessite une **pratique régulière et assidue** de votre part.

1.3 ODL dans la formation

Logique d'organisation des cours du 1er semestre

La figure 1.1 illustre les relations entre les différents cours du premier semestre de la première année, soulignant leur complémentarité.

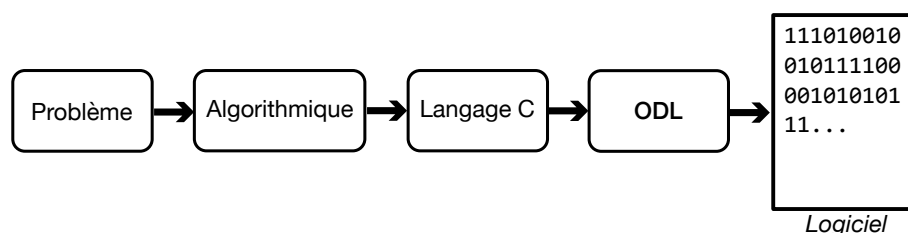


Fig. 1.1 : Logique d'organisation des cours.

Le cours d'algorithmique permet aux étudiants de modéliser conceptuellement une solution informatisable. Le langage C, enseigné dans un autre cours, est utilisé pour l'implémentation de ces solutions. Le cours d'outils de développement logiciel se concentre sur le codage sur la machine afin de produire des logiciels exécutables. Enfin, le cours sur l'architecture des ordinateurs, qui n'est pas représenté sur la figure, fournit des éléments sur l'environnement matériel, ainsi que la structure et l'exécution des programmes.

Formation morale

Les cours du premier semestre posent les bases de l'informatique. Ils exigent donc une **pratique régulière et intense**. Bien qu'il y ait relativement peu de cours en première année, ils demandent un investissement personnel

considérable, nécessitant maturité, autonomie et responsabilité. Le premier semestre est donc **primordial** pour la réussite future des étudiants, et il est donc essentiel qu'ils s'y investissent pleinement.

1.4 Contenu et modalités

Contenu du cours

Le cours est décomposé en deux grandes parties :

1. Système et environnement de développement
 - Environnements informatiques.
 - Commandes Unix/Linux.
 - Langage de commandes *shell* (bash).
2. Outils de développements
 - Édition de code source (utilitaires, présentation, commentaires).
 - Compilation (exhaustif) et création de bibliothèques statiques et dynamiques.
 - Utilitaires rencontrés : *vim*, *doxygen*, *gcc*, *gdb*, *valgrind*, *make*, etc.

Modalités

Pour les élèves sous statut étudiant, le cours comprend seize heures de cours magistraux réparties sur neuf semaines, ainsi que onze séances de travaux pratiques de deux heures chacune. Quant aux élèves sous statut apprenti, le cours représente douze heures et les travaux pratiques 20 heures.

Bien que le langage C soit utilisé durant les TP, il n'est pas le sujet central des séances.

Examens

Les étudiants seront évalués à l'aide de deux notes :

- Une note basée sur les travaux pratiques et le contrôle continu (50 %).
- Une note issue de l'examen final (50 %).

2 Ordinateurs et systèmes d'exploitation

« Computers in the future may weigh no more than one-and-a-half tonnes. » Popular Mechanics, 1949.

2.1 Introduction

Un ordinateur est la raison d'être de l'informatique. Il constitue la base sur laquelle repose l'ensemble des technologies de l'information et de la communication.

Machine (Pascal 1642, Jacquard 1801, Babbage 1821)

Dans le cas d'une machine, c'est le programme qui fait la machine. Les décisions logiques sont prises par le constructeur, comme l'illustre la figure 2.1. Cela signifie que chaque machine est spécifiquement conçue pour effectuer une tâche particulière, et son fonctionnement est déterminé par sa construction physique.

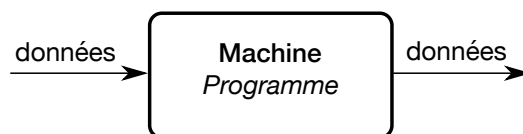


Fig. 2.1 : Une machine et son programme intégré.

Par exemple, en 1801, Joseph-Marie Jacquard a mis au point un métier à tisser automatisé dans lequel les données, sous forme de cartes perforées, étaient introduites automatiquement dans la machine. Cette invention a marqué une étape importante dans l'automatisation des tâches et a inspiré de futures innovations dans le domaine des machines programmables.

Ordinateur (ENIAC 1945)

En 1945, John von Neumann a défini l'architecture de l'ordinateur moderne, où le programme est considéré comme une « donnée » de l'ordinateur ainsi que le représente la figure 2.2. Contrairement aux machines précédentes, les décisions logiques sont maintenant prises par l'ordinateur lui-même, ce qui permet une flexibilité et une puissance de calcul accrues.

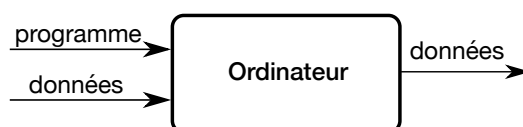


Fig. 2.2 : Un ordinateur.

Un ordinateur ne sait rien faire si ce n'est exécuter des instructions de bas niveau câblées dans un **microprocesseur**, comme l'addition, l'affectation et un compteur d'instructions. Il peut également charger un **programme** à partir d'un périphérique extérieur, géré par le BIOS (*Basic input output software*). Le BIOS est un logiciel embarqué qui initialise et teste le matériel du système lors du démarrage et charge le système d'exploitation.

2.2 Architecture matérielle et logicielle d'un ordinateur

Matériel (*hardware*)

L'architecture matérielle d'un ordinateur comprend principalement plusieurs composants essentiels :

- **Microprocesseur** : C'est le cerveau de l'ordinateur qui exécute les instructions des programmes. Les familles de processeurs regroupent ceux qui peuvent exécuter le même jeu d'instructions, comme les processeurs x86 (Intel, AMD), PowerPC (IBM), et SPARC (Sun).
- **Mémoire vive (RAM, *Random access memory*)** : Elle stocke temporairement les données et les instructions que le microprocesseur doit traiter rapidement.
- **Unité de stockage** : Les disques durs ou d'autres supports stockent les données et les programmes de manière permanente.

Avec l'évolution des technologies, la loi de Moore qui prédit que le nombre de transistors dans un microprocesseur double environ tous les deux ans, montre ses limites. Aujourd'hui, nous nous dirigeons vers des processeurs multicœurs, qui contiennent plusieurs unités de traitement. Ces processeurs sont adaptés aux systèmes d'exploitation multitâches, mais ils nécessitent que les programmes soient conçus pour le parallélisme afin d'exploiter pleinement leurs capacités.

La RAM est mesurée en octets (*bytes*) avec des préfixes tels que kilo (10^3), méga (10^6), giga (10^9), etc. Par exemple, une configuration d'ordinateur moderne pourrait inclure un processeur Intel Core i7, 16 Go de RAM et un disque dur SSD de 1 To.

Logiciel (*software*)

Le logiciel d'un ordinateur inclut plusieurs éléments clés :

- **BIOS** : Le BIOS est un *firmware* ou « logiciel embarqué » qui constitue le système d'exploitation minimum fourni par le constructeur. Il permet de lire et d'exécuter le contenu d'un CD-ROM ou d'une clé USB de démarrage (*bootable*) et de charger le système d'exploitation depuis le disque.
- **Système d'exploitation** : Le système d'exploitation est le logiciel principal qui permet d'exploiter les ressources de la machine. Il gère les fichiers, les protocoles réseau, les périphériques, les comptes utilisateurs et l'exécution des programmes.
- **Langage de commandes** : Le langage de commandes offre un accès aux fonctionnalités du système d'exploitation par l'intermédiaire de commandes prédéfinies. Il permet de réaliser des tâches administratives et de manipulation de fichiers directement à l'aide d'une interface en ligne de commande.
- **Interface graphique** : L'interface graphique permet de manipuler les fonctionnalités du système d'exploitation de manière visuelle en utilisant des fenêtres, des icônes et des menus.
- **Logiciels** : Les logiciels sont des programmes qui utilisent les fonctionnalités du système d'exploitation pour effectuer diverses tâches, comme le traitement de texte, la navigation sur Internet ou le montage vidéo.

2.3 Autour de l'unité centrale

L'unité centrale d'un ordinateur est le boîtier principal qui abrite les composants matériels de base. Elle est connectée à divers périphériques qui permettent l'interaction avec l'utilisateur et l'exécution de tâches spécifiques. Les périphériques courants incluent :

- Clavier et souris qui permettent l'entrée de données et la navigation.
- Écran qui affiche les informations et l'interface graphique.
- Carte son qui gère les sorties audio.
- Lecteurs CD-ROM/DVD et clés USB qui permettent l'entrée et la sortie de données à l'aide de supports amovibles.
- Appareil photo, Webcam, lecteur de cartes bancaires qui autorisent des interactions spécifiques.

Les **pilotes** (*drivers*) sont des programmes qui permettent au système d'exploitation de gérer ces périphériques, assurant leur bon fonctionnement et leur compatibilité avec le reste du système.

2.4 Environnements

Quatre environnements standards coexistent actuellement, chacun se distinguant par son système d'exploitation, lequel est chargé par le BIOS. Ces environnements sont :

Ms-Windows (1995)

- Distributeur : Microsoft (lié aux processeurs Intel).
- Système d'exploitation : Ms-DOS -- Ms-Windows.
- Langage de commandes : Ms-DOS (`command.com`, `cmd.exe`) et depuis Ms-Windows 10 : *Powershell* et *bash*.
- Bureau : Ms-Windows 95, 98, 2000, XP, Vista, 7, 8, 10.
- Histoire : 1981 rachat (50 000 dollars) et léger remaniement par William Gates et Paul Allen du système d'exploitation QDOS¹ pour IBM. Le succès d'IBM entraîne le succès du Ms-DOS.

Unix (1969)

- Distributeur : Sun (*Stanford University Network*) / OpenSolaris ; HP ; Université de Berkeley (BSD).
- Multitâches, multiutilisateurs, multiplateformes (Intel, SPARC, Motorola).
- Système d'exploitation : FreeBSD, OpenBSD, Solaris10.
- Langage de commandes : *shell* (*sh*, *bash*, *ksh*, *csh*, *tcsh*, *zsh*).
- Bureau : CDE, WindowMaker, KDE, Gnome.
- Histoire : 1975 AT&T par les inventeurs du langage C (Ken Thompson et Dennis Ritchie) -- en fait le C a été créé pour réaliser Unix.
- Au début gratuit, la fermeture du système par AT&T en 1980 permet à des projets tels que BSD, Hurd et Linux de voir le jour.

Linux (1991)

- Distributeurs : Red Hat / [Fedora](#), Canonical / [Ubuntu](#), Suse GmbH / [Opensuse](#), etc.
- Système d'exploitation : Linux.

¹QDOS signifie *quick-and-dirty operating system*, soit « système d'exploitation vite fait mal fait ».

- Langage de commandes : *shell* (bash, sh, csh, ksh, zsh, etc.).
- Bureau : KDE (Ms-Windows like), Gnome, Xfce, etc.
- Histoire : système d'exploitation proposé par Linus Torvalds, et basé sur [Minix](#), un « mini » Unix créé dans un but pédagogique par Andrew Tanenbaum [13].

Macintosh 🍏 (1984)

- Distributeur : Apple (logo : pomme de la connaissance).
- En plus, fabricant d'ordinateurs : MacBook, MacBook Pro, iMac.
- Système d'exploitation : Mac OS X basé sur le micronoyau Hurd et des composants du système FreeBSD (Darwin).
- Langage de commandes : *shell* (sh, bash, etc.).
- Bureau : Aqua.
- Histoire : Steve Jobs et Stephen Wozniak s'inspirent de l'interface graphique de Xerox (souris, fenêtres, etc.) pour créer LISA puis Macintosh.

Répartition des ventes

La figure 2.3 présente l'évolution du marché des systèmes d'exploitation vendus ou installés entre janvier 2009 et juillet 2024. Linux représente environ 1,5 % des parts de marché.

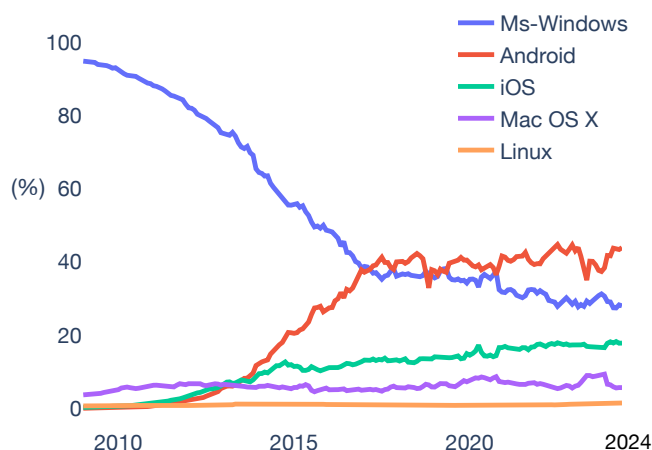


Fig. 2.3 : Marché des systèmes d'exploitation 2009-2024 (source : gs.statcounter.com).



Ces statistiques concernent principalement les ordinateurs personnels. Unix est utilisé sur les systèmes de calcul intensif, tandis que Linux et Mac OS X sont couramment trouvés sur les serveurs Internet et dans les domaines de l'édition et de la musique.



Plusieurs systèmes d'exploitation peuvent être installés sur une même machine à l'aide d'un amorçage multiple (*multiboot*), bien que la tendance actuelle soit plutôt d'utiliser la [virtualisation](#) (ex. : [Qemu](#), [VirtualBox](#)) ou la [conteneurisation](#) (ex. : [Docker](#)).

Comparatif des systèmes d'exploitation

Pour aider à comparer les différents systèmes d'exploitation, voici un tableau résumant leurs points positifs et négatifs :

Points positifs

Ms-Windows	Linux	Unix	Mac OS X
Le plus répandu	Gratuit	Gratuit (hum)	Logiciel : gratuit
<i>Plugins</i>	Logiciel : gratuit	Logiciel : gratuit	Ergonomie
Pilotes	Puissance	Robuste	Robuste
Accessoires	Sécurité	Puissance	Sécurité
Multiplateformes		Multiplateformes	<i>Leader</i> dans la création

Points négatifs

Ms-Windows	Linux	Unix	Mac OS X
Payant	Trop nombreuses	Trop nombreuses	Payant
Ergonomie	distributions	versions	Puissance
Puissance	<i>Plugins</i>	<i>Plugins</i>	Propriétaire
Sécurité	Pilotes	Pilotes	
« Racketiciels » (installés par défaut)	Accessoires	Accessoires	
Incompatibilité (peu soucieux des normes)			
Attaché aux processeurs Intel et ses clones			




























2.5 Logiciels

L'exécution d'un logiciel dépend du système d'exploitation et du microprocesseur. Il existe différentes typologies de logiciels :

- Logiciels payants : nécessitent l'achat d'une licence pour être utilisés.
- Partagiciels (*shareware*) : peuvent être utilisés gratuitement pendant une période d'essai, après quoi une licence doit être achetée.
- Gratuiciels (*freeware*) : sont gratuits, mais le code source n'est généralement pas disponible.
- Logiciels libres (*open source*) : sont gratuits et leur code source est disponible, permettant à quiconque de les modifier et de les distribuer.

Logiciel libre

L'inspirateur du mouvement des logiciels libres (*open source*)² est Richard Stallman qui a lancé le projet GNU dans les années 1980 au MIT. Le projet GNU a produit des outils essentiels comme *Emacs* et *gcc*. Les logiciels libres offrent des alternatives aux logiciels propriétaires dans divers domaines :

Logiciel propriétaire	Alternative libre	Systèmes d'exploitation
Suite Microsoft Office	LibreOffice	  
PhotoShop	Gimp	  
Adobe Illustrator	Inkscape, Krita	  
QuarkXPress	Scribus	  
Adobe Premiere, iMovie	Kdenlive	  
Internet Explorer	Firefox, Brave	  
Outlook express	Thunderbird	  
Windows Media Player	VLC	  
Winrar	7-zip	  

Une liste plus complète d'alternatives libres peut être trouvée sur le site framalibre.org

2.6 Accès aux informations



Moteurs de recherche

Pour trouver des informations en ligne, plusieurs moteurs de recherche sont disponibles, tels que :

- Searx : searx.space (liste des serveurs Searx)
- Qwant : lite.qwant.com
- Duck Duck Go : duckduckgo.com
-  : www.google.com

Distributions courantes de Linux



Il existe une centaine de distributions de Linux. En voici quelques-unes parmi les plus populaires :

-  : www.ubuntu-fr.org (débutant)
- Debian : www.debian.org (moyen)
-  : getfedora.org (débutant)
-  : software.opensuse.org (débutant)
- Manjaro : <https://manjaro.org> (débutant)
- Arch Linux : www.archlinux.org (initié)

²Pour plus de renseignements sur les alternatives du monde libre, allez voir la page [Wikipedia](#) correspondante.

Sites de référence de l'école

Quelques sites de référence utiles à l'école incluent :

- Site officiel : www.ensicaen.fr .
- My ENSICAEN : ensicaenfr.sharepoint.com (le concentrateur de toutes vos informations, mais un fouilli monumental).
- Vos cours : foad.ensicaen.fr (les cours et travaux pratiques en ligne).
- Votre scolarité : scolarite.ensicaen.fr (les emplois du temps).
-  : gitlab.ecole.ensicaen.fr (vos projets personnels).

Actualités informatiques

Pour suivre les actualités informatiques, des sites comme linux.slashdot.org sont recommandés.

Les incontournables

Quelques sites incontournables dans le domaine de l'informatique incluent :

- GNU : www.gnu.org
- Openclassrooms : openclassrooms.com
-  : stackoverflow.com
-  : github.com
-  : bitbucket.org
- SourceForge : sourceforge.net

3 Connexion

« Treat your password like your toothbrush. Don't let anybody else use it, and get a new one every six months.
» Clifford Stoll.

Dans ce chapitre nous fournissons quelques indications concernant les comptes utilisateurs à l'école.

3.1 Compte personnel

Systèmes de l'école

Les systèmes d'exploitation installés dans les salles machine sont Ms-Windows et Linux (distribution Ubuntu).

Compte personnel à l'école

Considérons l'utilisateur Jean Saigne. Celui-ci dispose à l'école de :

- Espace disque d'environ 500 Mo (à la fois Unix et Ms-Windows)
- Dossier personnel : `/home/jsaigne`
- Espace courriel (environ 300 Mo)
 - Courriel : `jsaigne@ecole.ensicaen.fr`
 - Accessible depuis <https://outlook.office.com> afin d'aider la NSA à mieux vous connaître.
- Accès à distance :
 - **sftp** : `ftp://ftp.ecole.ensicaen.fr` (transfert de fichiers)
 - **ssh** : `ssh://cybele.ecole.ensicaen.fr` (travail à distance -- mode terminal)
 - **Wifi** : voir le didacticiel sur [My ENSICAEN](#)

Vous devez entre autres apprendre à gérer votre compte et à faire de la place périodiquement afin de préserver votre espace disque (par exemple en vidant le dossier d'envoi de courriers électroniques, le cache du navigateur, etc.).



La gestion d'un système Unix / Linux est réalisée par un « administrateur » qui possède **tous les droits**. Cet administrateur s'appelle **super-utilisateur** ou encore utilisateur *root*. Il peut entre autres configurer le système, installer et désinstaller des logiciels, donner ou retirer des droits, et éventuellement **surveiller vos actions** et **accéder à vos fichiers**. Les autres utilisateurs disposent de droits limités vis-à-vis du système, et de droits complets en ce qui concerne leurs comptes personnels pour peu qu'ils respectent la charte d'utilisation.

3.2 Connexion : mot de passe

3.2.1 Mots de passe : les clés du système

Avant de pouvoir utiliser un système multi-utilisateur, il faut se connecter à l'aide d'un identificateur (*username* ou *login*) et d'un mot de passe (*password*).

- Identification : identificateur
- Authentification : mot de passe

Ces deux éléments combinés avec le dossier d'accueil représentent le point d'entrée ou « compte personnel » (*account*) d'une personne sur le système.

C'est le mot de passe qui est important, et non pas le code d'identification qui est public.

Le mot de passe est *inaccessible*.

3.2.2 Conséquences du vol de mot de passe

Lorsqu'un intrus réussit à obtenir votre mot de passe, il peut :

- consulter vos données personnelles, les modifier voire les supprimer;
- se faire passer pour vous afin d'effectuer des actes répréhensibles, dont la responsabilité vous incomberait;
- abuser des privilèges qui vous ont été accordés;
- occuper votre espace disque pour y mettre des fichiers interdits (ex. : logiciels piratés).



Il est de votre devoir de tout faire pour garantir l'intégrité de votre mot de passe (cf. charte + cf. ce qui suit). Vous êtes responsable de l'espace disque que l'on vous a octroyé.

3.2.3 Moyens de voler un mot de passe

Pour un mot de passe de huit caractères, il existe 128 combinaisons. Il est impossible de les essayer toutes, mais il existe des méthodes permettant de percer à jour les mots de passe.

Piratage

Les logiciels de piratage vont permettre l'intrusion sur un système, notamment les *crackers* à base de dictionnaires qui se trouvent facilement sur le Darknet. Les mots du dictionnaire sont testés à l'endroit, à l'envers, ou encore entrecoupés de caractères tels que le tiret bas « _ ».



Attention toute utilisation d'un *cracker* de mot de passe à l'école entraîne des sanctions disciplinaires!



Concernant les personnes à même de s'introduire sur un système, attention à la différence entre *hackers* (connaît tout sur l'informatique) et *crakers* (délinquant).

Reniflage

Un renifleur (*sniffer*) est un programme basé sur l'écoute du réseau. Il suppose que les mots de passe sont passés en clair. La solution consiste à utiliser des protocoles cryptés à base de clés, du type *ssh* ou *shtml*.

Hameçonnage

L'hameçonnage ou appâtage (*phishing* pour *password harvesting fishing*, soit « pêche aux mots de passe ») est une méthode basée sur la naïveté. Il s'agit de se faire passer pour une autorité. Par exemple : envoyer un *mail* en se faisant passer pour l'autorité et demander à ce que vous changiez votre mot de passe pour un autre qui serait fourni.* Le plus connu des hameçonnages consiste à imiter l'écran de connexion.

Espionnage

L'espionnage utilise différentes techniques, comme l'observation des mains en plaçant par exemple une caméra au-dessus du clavier, la fouille des poubelles, ou encore l'analyse des *post-its* autour de l'écran.

Cheval de Troie

Un cheval de Troie est un programme introduit sur la machine hôte et qui espionne cette dernière.



La création et l'utilisation de faux écrans entraînent des sanctions* disciplinaires (cf. charte)!

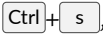
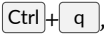
3.2.4 Comment choisir les bons mots de passe ?

Un mot de passe sur un système Unix, c'est huit caractères au minimum et douze au maximum, avec au moins un caractère non alphabétique (numérique ou spécial), mais au moins deux caractères alphabétiques.

Règle n°1

Vérifiez que votre mot de passe n'appartient à aucun dictionnaire [*crackers*] noms communs, noms propres, calendrier, marques, dans n'importe quelle langue, à l'endroit et à l'envers, augmenté d'un caractère ou de chiffres. Exemples : *p@ssw0rd*, *_g0ldf1sh*.

Règle n°2

Assurez-vous que votre mot de passe mélange des caractères alphanumériques et des caractères spéciaux. Parmi les caractères non alphabétiques, évitez les caractères d'effacement par défaut (vérifiez-le avec les commandes « *stty -a* », « *stty all* », selon les systèmes). Attention aux séquences de touches telles que , , etc. Préférez-leur des caractères imprimables comme le tiret bas « *_* » ou encore des chiffres.

Règle n°3

Vérifiez que votre mot de passe est indépendant de la machine sur laquelle il est tapé en excluant les caractères de contrôle ou les caractères spécifiques de clavier (« *ç* », « *ü* », etc.).

Règle n°4

Utilisez des moyens mnémotechniques afin d'éviter des traces écrites qui faciliteraient l'espionnage. Si possible, n'inscrivez jamais un mot de passe sur un papier, ou alors modifiez-le pour qu'il semble être toute autre chose.

Il existe des utilitaires qui permettent de générer ou de valider des mots de passe (nous verrons un script *shell* plus loin qui peut générer des mots de passe valides).

Ce qui rend les mots de passe faciles à mémoriser est l'utilisation de termes mnémoniques qui associent un modèle de lettres et de nombres à des expressions qui vous sont familières.

Exemples de procédures mnémotechniques

Méthode n°1 La technique du SMS utilise les premières lettres d'une phrase mnémotechnique. Par exemple, la phrase « Il était une fois trois petits cochons... » devient « **ie1x3pc** » qui peut alors être utilisé comme un bon mot de passe. Vous pouvez aussi varier les majuscules et les minuscules.

Méthode n°2 La technique de substitution de voyelles nécessite d'apprendre par coeur une chaîne de caractères, puis de remplacer dans un mot choisi chaque voyelle par les caractères de la chaîne. Par exemple : la chaîne « **_*+?\$?£** » et le mot « **ensicaen** » donnera la chaîne chiffrée « ***ns+c_*n** ».

3.2.5 Changement de mot de passe

- Changez souvent de mot de passe, tous les deux mois par exemple.
- Utilisez des mots de passe différents sur tous les systèmes. Choisissez plutôt un bon squelette de mot de passe et ajoutez-y un préfixe ou un suffixe comme identificateur pour chaque système.
- Changez de mot de passe quand vous pensez qu'on l'a pénétré (ex. : votre mot de passe s'est affiché à l'écran à la suite d'une erreur de manipulation).
- Si vous oubliez votre mot de passe, le super-utilisateur peut en remettre un nouveau. À n'utiliser qu'avec parcimonie à l'école !

3.2.6 Comment abandonner votre ordinateur ?

Ne laissez jamais votre ordinateur sans surveillance. Déconnectez-vous ou utilisez un programme de verrouillage ¹ approuvé pour protéger votre identifiant d'utilisateur.

3.3 Ouverture d'une session Ubuntu

Une fois connecté (identifié et authentifié), vous entrez dans l'environnement graphique **Gnome** ².

Dans les cours de première année, votre fenêtre de travail courante sera la plupart de temps un **terminal**. À l'intérieur de ce terminal, un interpréteur de commandes ou *shell* s'exécute. Par défaut, c'est l'interpréteur appelé *bash*. L'interpréteur de commandes permet d'accéder aux fonctionnalités du système d'exploitation sous la forme de commandes référencées : gestion des entrées et sorties, montage de périphériques, lancement de programmes, etc.

¹Un programme de verrouillage utilise le mot de passe que vous avez utilisé quand vous vous êtes connecté, cela afin de verrouiller la ligne et le terminal de communication. Vous devez à nouveau entrer le mot de passe correct afin de déverrouiller la ligne et le terminal.

²Gnome est totalement écrit en langage C, même s'il adopte une approche objet dans son implémentation !

4 Organisation des disques

« I think Microsoft named .Net so it wouldn't show up in a Unix directory listing. » Otkar.

Dans ce chapitre, nous décrivons la structure du contenu d'un disque dur pour un système Unix. Nous y abordons les notions d'arborescence, de dossiers et de fichiers.

4.1 Structure arborescente des disques

Les systèmes de fichiers sont organisés hiérarchiquement sous la forme d'**arbres**. Un arbre est constitué de **noeuds** (dossiers) et de **feuilles** (fichiers).

Sur les systèmes d'exploitation Unix, Linux et Mac OS X, il n'y a qu'un seul arbre dont la racine est nommée « / » comme l'illustre la figure 4.1. Dans le cas du système Ms-Windows, il s'agit d'une « forêt » constituée d'un ensemble d'arbres associés aux différents systèmes de fichiers installés et dont les racines sont nommées par des lettres (« C: », « D: », « E: », etc.).

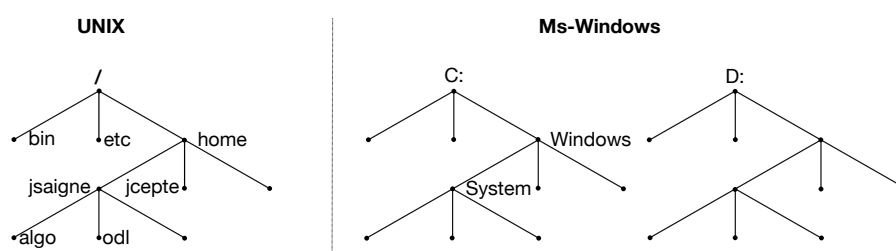


Fig. 4.1: Arborescences sur Unix et MsWindows.

Un dossier contient des fichiers et d'autres dossiers, leur nombre étant limité par la taille du disque. Par exemple : « bin », « dev » sont des dossiers présents à la racine d'un système Unix.

Un fichier, quant à lui, contient les données d'une application quelconque. Par exemple, le fichier « battle.c » est un fichier contenant par convention du code écrit en langage C.



Sous Unix, chaque disque s'ajoute comme une nouvelle branche de l'arbre. Par exemple, une clé USB sera en général sous Linux, « montée » dans le dossier « /media/nom_du_volume ».

4.2 Dossier ou répertoire

À chaque instant dans un terminal, il n'y a qu'un dossier (*folder*), appelé aussi répertoire (*directory*), d'ouvert : le dossier courant que l'on appelle aussi « répertoire de travail » (*working directory*). Tout ce qui est créé sans spécifier le nom d'un dossier se fait dans ce dossier.

4.2.1 Nom de dossier

Un dossier est identifié par un nom qui est une suite de caractères sans espace ni signe de ponctuation et de taille inférieure à 254 caractères. Dans le cas contraire, chaque caractère espace ou de ponctuation devra être précédé du caractère de désépéciation « \ ». Cela est donc à éviter.

Il existe des dossiers particuliers :

- le dossier parent « . . », ce qui évite de connaître son nom;
- le dossier lui-même « . » ce qui évite aussi de connaître son nom;
- le dossier racine d'un utilisateur : \$HOME ou « ~ » ou encore « ~identifiant ».

4.2.2 Chemin absolu

Un chemin absolu (*absolute pathname*) indique la liste des noms des dossiers depuis la racine, séparés par le caractère « / ».

Par exemple : « /home/jsaigne/odl/tp1/exercice1 ».

4.2.3 Chemin relatif

Un chemin relatif (*relative pathname*) est décrit par rapport au dossier courant. Par exemple, si le dossier courant est « /home/jsaigne/ », alors le chemin relatif du sous-dossier « exercice1 » depuis le dossier courant sera : « ./odl/tp1/exercice1 » ou « odl/tp1/exercice1 ».

4.2.4 Dossiers de base

Les dossiers présents à la racine d'un système Unix sont généralement :

- **/bin** : utilitaires Unix de base;
- **/dev** : fichiers périphériques (l'entrée standard `/dev/stdin`, la sortie standard `/dev/stdout`, le « trou noir » `/dev/null`, etc.);
- **/etc** : programmes d'administration du système et tables diverses;
- **/home** : dossiers personnels des utilisateurs;
- **/lib** : bibliothèques associées aux programmes présents dans `/bin`;
- **/media** : disques externes, clés USB, etc.;
- **/usr** : programmes utilisateurs en général;
- **/usr/bin** : autres utilitaires plutôt dédiés utilisateurs (ex. : *vim*, *firefox*, etc.);
- **/usr/local** : programmes rapatriés par l'utilisateur;
- **/usr/lib** : autres bibliothèques plutôt dédiées utilisateurs;
- **/tmp** : fichiers temporaires;
- etc.

4.3 Fichier

4.3.1 Types de fichiers

On rencontre principalement trois types de fichiers ¹ :

¹En réalité, il existe aussi des fichiers dits *spéciaux* que nous traiterons dans le cours de systèmes d'exploitation en 2e année.

1. fichier de données texte;
2. fichier de données binaire;
3. fichier exécutable (logiciel).

4.3.2 Noms de fichiers

Un fichier est identifié par un nom de taille inférieure à 254 caractères. Ce nom est formé d'un radical et d'un suffixe (optionnel sous Unix / Linux). Le suffixe est généralement lié au type du fichier (p. ex. « .c » pour un fichier écrit en langage C), mais cela reste arbitraire.

4.3.3 Fichier de données texte

Un fichier texte peut être visualisé à l'aide d'applications graphiques ou encore à l'aide de commandes Unix.

Les suffixes rencontrés seront par exemple :

- « .c », « .cc », « .php », « .java », etc. pour les fichiers de programmation;
- « .txt », « .md », etc. pour les fichiers textes (voire un radical en majuscules sans suffixe).

4.3.4 Fichier de données binaire

Un fichier de données binaire nécessite l'utilisation d'un logiciel adapté pour sa manipulation. Quelques exemples :

- « .ps », « .pdf », « .rtf », « .doc » : les logiciels ghostview, acroread (*Acrobat reader*), pandoc, etc.
- « .jpg », « .png », « .tiff », « .gif » : les logiciels convert, gimp, etc.

4.3.5 Fichier exécutable

L'exécution d'un fichier exécutable consiste en général à cliquer sur son icône depuis le gestionnaire de fichier si cet exécutable met en oeuvre une interface graphique, ou encore à appeler son nom depuis un terminal en indiquant le chemin vers celui-ci.

Il existe un certain nombre de chemins connus et dans lesquels le système va chercher les programmes. La variable PATH contient la liste des dossiers où sont stockés des fichiers exécutables de type « logiciel ». Le premier trouvé est exécuté.

4.3.6 Fichier caché

Un fichier caché commence par un point. Par exemple, le fichier « .profile » que nous rencontrerons plus loin est un fichier caché.

4.4 Protection des dossiers et fichiers

Unix distingue trois groupes d'utilisateurs :

- u (*user*) : l'utilisateur;
- g (*group*) : les groupes auxquels appartient l'utilisateur;

- **o** (*other*) : l'ensemble des utilisateurs hors des groupes ;
- **a** (*all*) : l'ensemble des utilisateurs.

Les fichiers et dossiers appartiennent à un propriétaire (en général l'utilisateur) et à au moins un groupe (en principe celui auquel est affilié l'utilisateur).

Les droits d'accès aux fichiers et dossiers sur Unix sont les suivants : **r**, **w**, **x**. Suivant leur type (dossier ou fichier), les droits d'accès correspondent à :

- dossier :
 - **r** : possibilité de visualiser le contenu.
 - **w** : possibilité de créer ou modifier un dossier ou un fichier à l'intérieur.
 - **x** : possibilité d'ouvrir le dossier (c'est-à-dire pouvoir faire « `cd dossier` »).
- fichier :
 - **r** : possibilité de lire le contenu du fichier.
 - **w** : possibilité de modifier le fichier.
 - **x** : possibilité d'exécuter le fichier.

Nous verrons dans le chapitre suivant que la commande Unix `ls` permet de visualiser les informations sur les fichiers et dossiers de même que leurs droits d'accès, que les commandes `chmod` et `umask` permettent de modifier ces droits d'accès, et que les commandes `chown` et `chgrp` permettent de modifier le propriétaire et le groupe auxquels ils appartiennent.

5 Commandes Unix

« Forget UNIX -- it will be gone in 5 years. » Thomas Jermoluk in the late 80's (Silicon Graphics Inc.).

Dans ce chapitre très dense, nous présentons des commandes de base d'un système Unix. Le lecteur curieux pourra se reporter dans un premier temps à l'ouvrage de J.-P. Armspach, puis à ceux de B. Ward et C. Albing lorsqu'il souhaitera approfondir ses connaissances sur le sujet [2,3,14]. Toutefois, rien ne remplacera une pratique intense de ces différentes commandes afin de maîtriser, puis d'apprécier leur utilisation.

5.1 Introduction

Bien qu'il soit possible d'utiliser directement l'environnement graphique afin d'effectuer la plupart des opérations sur une machine, nous n'utiliserons la plupart du temps qu'un terminal dans lequel nous entrerons des commandes -- Unix en fournissant plus de 150 de base.

L'avantage de cette démarche est triple :

- les commandes permettent d'accéder à la totalité des fonctionnalités d'un système (les applications graphiques sont parfois trop simplifiées et bridées ou trop complexes et lourdes);
- la faible empreinte mémoire de ces commandes est en accord avec une démarche en développement durable;
- Au premier semestre, comme vous n'allez développer que des applications « textuelles » qui s'exécuteront dans un terminal, autant apprendre à le maîtriser.

Conventions de notation

Nous allons rencontrer deux types de commandes auxquelles on accédera depuis un terminal :

1. **\$ commande** : commande à taper depuis l'invite de l'interpréteur de commandes (ici l'invite, appelée aussi « prompt », est représentée par le caractère « \$ » -- nous ne le ferons d'ailleurs pas apparaître lorsque la commande est insérée dans un paragraphe, seulement dans un bloc de code). Exemple : « \$ vim file.c » (ou simplement « vim file.c »).
2. **logiciel> commande** : commande à taper dans l'interpréteur de commandes d'un logiciel spécifique. Par exemple : « gdb> run [arguments] ».

5.1.1 Qu'est-ce qu'une commande Unix ? {-}

Toute commande Unix peut être vue comme une boîte noire avec une entrée et deux sorties, ainsi que le représente la figure 5.1.

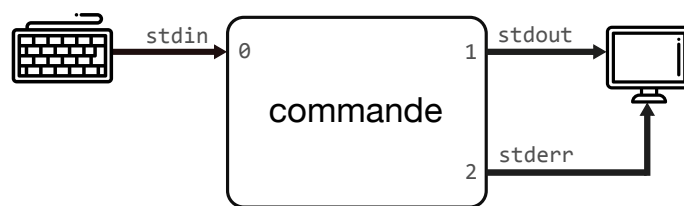


Fig. 5.1 : Représentation d'une commande Unix avec ses entrées-sorties par défaut.

Les numéros associés aux entrées et sorties sont appelés **descripteurs de fichiers**. Ce sont des valeurs entières attribuées par le système d'exploitation, et elles sont constantes pour toute commande ou programme. Le descripteur **0** représente l'entrée standard (fichier spécial de flux d'entrée de caractères `stdin`), qui est par défaut associé au clavier. Le descripteur **1** représente la sortie standard (`stdout`), généralement l'écran. Le descripteur **2** est utilisé pour la sortie des erreurs (`stderr`), également dirigée vers l'écran par défaut.

Unix a été conçu selon la philosophie que **tout est fichier**. Les dossiers, les gestionnaires de périphériques, et même les entrées et sorties de commandes ou programmes sont traités comme des fichiers. Cette approche permet de rediriger facilement les entrées et sorties vers d'autres fichiers que les gestionnaires matériels par défaut. Elle permet également de chaîner plusieurs commandes, de manière à créer des pipelines complexes. La figure [figure ?] illustre un pipeline constitué de trois commandes Unix.

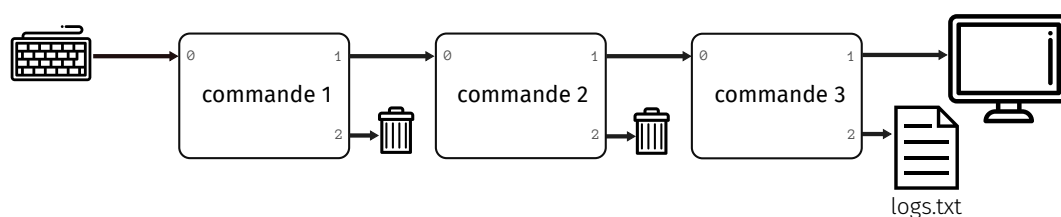


Fig. 5.2 : Exemple d'un pipeline de commandes.

À propos du terminal

Voici tout d'abord quelques règles à retenir lorsqu'on utilise un terminal :

- dans le monde Unix, majuscules et minuscules sont différenciées (les commandes sont généralement en minuscules);
- dans le terminal, une commande n'est pas soumise tant que l'on n'a pas appuyé sur la touche `Enter`;
- l'interpréteur de commandes dispose d'un mécanisme de complétion des commandes à l'aide de la touche de tabulation `␣`;
- afin de corriger un des éléments de la commande en déplaçant le curseur, il est possible d'utiliser les touches `←` et `→`; de la même façon, les raccourcis `Ctrl + a` et `Ctrl + e` permettent de placer le curseur respectivement en début et en fin de ligne, `Ctrl + w` efface le mot précédent, `Ctrl + k` efface tout ce qui suit le curseur sur la ligne, `Ctrl + u` efface l'ensemble de la ligne et `Ctrl + c` abandonne la commande en cours;
- un historique des commandes préalablement exécutées est accessible à l'aide de la touche `↑` (parcours inverse avec `↓`) -- ou encore à l'aide de la commande `shell history` comme nous le verrons plus loin;
- le raccourci `Ctrl + l` efface le contenu affiché sur le terminal -- la commande `clear` réalise la même chose.

5.2 Documentation

Unix dispose d'un très grand nombre de commandes, de programmes de base, parfois avec de nombreuses options, ainsi que d'une interface de programmation conséquente. Il en résulte de très nombreuses informations à mémoriser. La commande `man` permet la consultation des pages de manuel concernant la plus grande partie de ces informations (commandes et options).

Les pages sont organisées en sections. On retiendra plus particulièrement les sections :

- **1** : programmes exécutables ou commandes de l'interpréteur ;
- **2** : appels système -- fonctions fournies par le noyau ;
- **3** : appels de bibliothèques -- fonctions fournies par les bibliothèques des programmes.

La syntaxe de `man` est la suivante :

```
$ man [options] [section] commande
```

Exemples

- `man man` : la page de manuel de `man`.
- `man 3 printf` : la fonction `printf()` de la bibliothèque standard du C.
- `man 1 printf` : la commande Unix `printf`.
- `man -a printf` : toutes les pages `printf.x` s'il y en a plusieurs.



La variable d'environnement `MANPATH` renseigne les dossiers dans lesquels la commande `man` va aller chercher les pages de manuel.

Autres commandes utiles

- **`apropos mot`** : cherche toutes les commandes qui se réfèrent à `mot` ;
- **`info commande`** : obtenir le manuel d'une commande (spécifique GNU).

5.3 Gestion des dossiers et des fichiers

5.3.1 Manipulation des dossiers

En ce qui concerne les dossiers, on utilisera très fréquemment les commandes suivantes :

Commande	Description
<code>cd</code>	déplacement du dossier courant vers un autre
<code>du</code>	donne la taille d'un dossier en octets
<code>ls</code>	affiche le contenu d'un dossier et renseigne sur les droits d'accès aux fichiers et dossiers
<code>mkdir</code>	création d'un dossier, mais sans l'ouvrir
<code>pwd</code>	affiche le chemin absolu du dossier courant

Commande	Description
<code>rmdir</code>	destruction d'un dossier (doit être vide)

Visualisation des droits d'accès

Les droits d'un dossier ou d'un fichier peuvent être visualisés à l'aide de la commande « `ls -l` ». Il est aussi possible d'utiliser « `ls -la` » (ou « `ls -lat` », a pour tous les fichiers y compris ceux qui sont cachés, et `t` pour la date du fichier) ou « `ls -ld` » pour se limiter aux dossiers.

```
$ ls -al
droits      liens  propriétaire  groupe  taille  date      nom
drwxr-xr-x  2      jsaigne        prof    340     Jui 26 17:32 .
drwxr-xr-x  6      jsaigne        prof    204     Jui 26 10:39 ..
-rw-r--r--  1      jsaigne        prof    7531    Feb 06 13:24 qmotor.c
```

Le champ de gauche indique les droits d'accès aux fichiers ou dossiers. Le premier caractère renseigne sur le type : « - » pour un fichier normal, « d » pour un dossier, « l » pour un lien, « p » pour un tube. Les suivants, par groupes de trois, renseignent sur les droits de l'utilisateur, du groupe et des autres. Viennent ensuite : le nombre de liens sur le fichier (ou le nombre de fichiers pour un dossier), le nom du propriétaire (en principe l'utilisateur, mais pas que...), le groupe, la taille en octets, la date et le nom.

Par exemple, le fichier « `qmotor.c` » ci-dessus est un fichier texte d'environ 7 Ko qui appartenant à l'utilisateur « `jsaigne` » et qui n'est modifiable que par celui-ci. On peut ajouter qu'il a été modifié pour la dernière fois le 6 février de l'année à 13h24, et qu'il n'a aucun lien pointant vers lui (cf. commande `ln`).

Exemples

- `cd ..` : retourne dans le dossier parent.
- `cd` ou `cd ~` : retourne dans le dossier personnel.
- `cd /usr/lib` : se déplace dans `usr` puis dans `lib` depuis la racine (quel que soit le dossier où l'on se trouve).
- `du -h od1/tp1` : affiche la taille du dossier en utilisant un multiple adapté de l'unité.
- `ls` : liste le dossier courant.
- `ls -l` : liste le dossier courant de façon détaillée.
- `ls -al od1/tp1` : liste le sous-dossier `tp1` de façon très détaillée en affichant aussi les fichiers et dossiers cachés.
- `ls -lha` : liste le dossier courant de façon très détaillée en affichant aussi les fichiers et dossiers cachés, et en utilisant si nécessaire, les multiples de l'unité afin d'indiquer les tailles.
- `mkdir algo algo/tp2` : crée le dossier `algo` et son sous-dossier `tp2`.
- `mkdir -p algo/tp2` : même chose.
- `cd od1 ; pwd` : affiche le chemin absolu du dossier `od1`.
- `rmdir od1/tp1` : supprime le sous-dossier `tp1` s'il est vide.

5.3.2 Manipulation des fichiers

Les commandes suivantes sont propres à la gestion des fichiers :

Commande	Description
<code>dd</code>	conversion et copie de fichiers
<code>file</code>	indique le type d'un fichier en se basant sur son entête
<code>iconv</code>	convertit l'encodage d'un fichier
<code>more</code>	affiche le contenu d'un fichier texte page par page (options : h, b, !, q, n)
<code>less</code>	affiche le contenu d'un fichier texte, avec une progression ligne par ligne ou page par page, ainsi que la possibilité de retourner en arrière et d'effectuer une recherche (spécifique à GNU)
<code>touch</code>	actualise la date du fichier et le crée s'il n'existe pas

Exemples

- `sudo dd if=~/.Download/ubuntu-20.04.iso of=/dev/sdb` : crée une clé bootable USB (supposée formattée et attachée dans « /dev/sdb ») à partir de l'image ISO récupérée dans le dossier de téléchargement.
- `file fic4.png` : retourne une information sur le type du fichier, par exemple « fic4.png : PNG image data, 234 x 292, 8-bit/color RGBA, non-interlaced ».
- `iconv -f ISO-8859-1 -t UTF-8 Ingles-2006.txt -o Ingles-2006.txt.utf` : convertit l'encodage du fichier en UTF-8 et le renomme.
- `touch fic.txt` : crée le fichier « fic.txt » s'il n'existait pas (taille nulle).
- `touch -t 1903200958 fic.txt` : modifie la date de dernière modification du fichier « fic.txt » au 20 mars 2019 à 9h58.

5.3.3 Commandes communes aux fichiers et dossiers

Les commandes suivantes sont communes à la gestion des dossiers et des fichiers :

Commande	Description
<code>basename</code>	récupère le nom d'un fichier sans le chemin et éventuellement sans le suffixe
<code>chgrp</code>	modifie le groupe auquel appartient un dossier ou un fichier
<code>chmod</code>	modifie les droits d'accès à un dossier ou un fichier
<code>chown</code>	modifie le propriétaire d'un fichier
<code>cp</code>	copie des fichiers et des dossiers
<code>dirname</code>	récupère le nom du dossier contenant le fichier
<code>find</code>	recherche récursive de fichiers ou dossiers, et éventuellement exécute une action dessus
<code>ln</code>	crée des « liens » entre des fichiers
<code>mv</code>	déplace ou renomme des fichiers
<code>rm</code>	efface des fichiers et des dossiers

Modification des droits d'accès avec `chmod`

La commande `chmod` (*change mode*) permet de changer les droits d'accès à un fichier ou un dossier. Sa syntaxe est la suivante :

```
$ chmod droits fichier(s)
```

On ajoute un droit à l'aide du caractère « + » et on le retire à l'aide du caractère « - », pour l'utilisateur (u), le groupe (g) ou les autres (o). Ainsi, la commande « `chmod +x fichier` » rend exécutable un fichier (comme un script, par exemple).

Une autre manière de modifier les droits est d'attribuer un chiffre correspondant aux droits recherchés comme suit :

- pour l'utilisateur, les droits d'accès en lecture sont de **400**, en écriture de **200** et en exécution de **100**.
- pour le groupe, les droits d'accès en lecture sont de **40**, en écriture de **20** et en exécution de **10**.
- pour les autres, les droits d'accès en lecture sont de **4**, en écriture de **2** et en exécution de **1**.
- on additionne ensuite les droits pour chacun.

Ainsi les droits « `rwxr-xr--` » sont équivalents à :

- **400 + 200 + 100 = 700** pour l'utilisateur (rwx)
- **40 + 10 = 50** pour le groupe (r-x)
- **4** pour les autres (r--)

soit au total **700 + 50 + 4 = 754**. On exécutera donc la commande « `chmod 754 fichier` ».

Exemples

- `basename /tmp/project1243/exo.c` : retourne « `exo.c` ».
- `basename /tmp/project1243/exo.c .c` : retourne « `exo` ».
- `for i in *; do; mv $i $(basename $i .gif).jpg; done` : modifie l'extension de tous les fichiers « `.gif` » du dossier courant en « `.jpg` ».
- `chmod 0700 odl` : rend le dossier `odl` inaccessible hormis à son propriétaire.
- `chmod +x fichier` : ajoute à tous les utilisateurs des droits d'exécution à un fichier, un script par exemple (+x est équivalent à « `a+x` »).
- `chmod g-w fichier` : retire les droits d'écriture aux autres membres du groupe.
- `chmod ug+x fichier` : rend le fichier exécutable pour l'utilisateur et le groupe.
- `cp f1 f2` : copie le fichier `f1` dans `f2`.
- `cp f1 f2 f3 f4 f5 d1` : copie les fichiers `f1` à `f5` dans le dossier `d1`.
- `cp -R d1 d2 d3 d4 d5 dest` copie les dossiers `d1` à `d5` dans le dossier `dest`.
- `dirname /tmp/project1243/exo.c` : retourne « `/tmp/project1243` ».
- `find ~/ -name "*fuzzy*" -print` : cherche et affiche tous les fichiers du dossier personnel dont le nom contient la chaîne « `fuzzy` ».
- `find . -exec grep "chaîne" {} \; -print` : cherche et affiche tous les fichiers du dossier courant qui contiennent « `chaîne` ».
- `find . -name "*.o" -exec rm {} \; -print` : cherche, affiche et supprime tous les fichiers d'extension « `.o` » du dossier courant.
- `find ~/ -size 450M -print` : cherche et affiche tous les fichiers du dossier personnel qui ont une taille supérieure à 450 Mio.
- `mv f1 f2` : renomme le fichier `f1` en `f2`.
- `mv f1 f2 f3 f4 f5 d1` : déplace les fichiers `f1` à `f5` dans le dossier `d1`.
- `mv d1 d2` : renomme le dossier `d1` en `d2`.
- `mv oldfile newfile` : renomme `oldfile` en `newfile`.
- `rm -ri odl/tp1` : supprime le dossier `tp1` que celui-ci soit vide ou non.

5.3.4 Manipulation de texte

La manipulation de texte peut être réalisée avec les commandes ci-dessous :

Commande	Description
<code>cat</code>	concatène des fichiers textes, mais permet aussi d'afficher le contenu d'un fichier
<code>cut</code>	supprime une partie des lignes d'un fichier selon un critère
<code>diff</code>	affiche les différences entre deux fichiers
<code>dos2unix</code>	convertit un fichier texte Ms-Dos dans son format Unix (sans retour chariot)
<code>grep</code>	recherche si la chaîne décrite par un motif est présente dans un(des) fichier(s)
<code>head</code>	affiche les <i>x</i> premières lignes (par défaut 10)
<code>sort</code>	trie les lignes de fichiers selon l'ordre alphabétique ou numérique
<code>tail</code>	affiche les <i>x</i> dernières lignes du fichier (par défaut 10)
<code>tee</code>	copie de l'entrée standard sur la sortie standard et vers un fichier
<code>tr</code>	transpose ou élimine des caractères dans une chaîne de caractères
<code>unix2dos</code>	convertit un fichier texte Unix dans son format Ms-Dos
<code>wc</code>	compte le nombre de mots, de lignes, de caractères, d'octets

Exemples

- `cat rouge.md` : affiche le contenu du fichier « `rouge.md` ».
- `cat rouge.md noir.md > rouge_et_noir.md` : concatène les fichiers « `rouge.md` » et « `noir.md` » et envoie le résultat dans le fichier « `rouge_et_noir.md` ».
- `cat rouge.md | wc -l` affiche le nombre de lignes du fichier « `rouge.md` ».
- `cat data.csv | cut -d; -f5` : récupère la 5e colonne du fichier « `data.csv` » dont les champs sont séparés par le caractère « `;` ».
- `grep "fuzzy" *.c` : cherche et affiche les lignes dans les fichiers « `.c` » du dossier courant qui contiennent le motif « `fuzzy.h` ».
- `ps -ed | grep jsaigue` : cherche et affiche les processus qui appartiennent à l'utilisateur « `jsaigue` ».
- `egrep expression_régulière fichiers` (variante de `grep` gérant les expressions régulières étendues).
- `head -5 rouge.md` : affiche les 5 premières lignes du fichier « `rouge.md` ».
- `tail -12 noir.md` : affiche les 12 dernières lignes du fichier « `noir.md` ».
- `ls | tee fichier` : affiche le contenu du dossier courant tout en ajoutant ces informations dans le fichier (pratique pour garder un œil sur la sortie standard tout en sauvegardant l'information).
- `unix2dos -ascii ../NEWS.txt \${DOSDIR}/NEWS.txt`.



La commande `cat` permet aussi d'éditer un nouveau fichier. Par exemple, on entre :

```
$ cat > mon_nouveau_rapport.txt
```

On écrit alors le contenu du fichier texte et on termine l'édition à l'aide de la séquence de touches `Ctrl+d`. Il est aussi possible de réaliser cette édition en entrant :

```
$ cat > mon_nouveau_rapport.txt << FINDEMONFICHIER
```

On écrit alors le contenu du fichier texte et on termine en réécrivant la chaîne de caractères (FINDEMONFICHIER). Cette dernière solution permet une utilisation dans des scripts *shell*.

Complément sur la commande `diff`

La commande `diff` indique les modifications à apporter au premier fichier passé en argument afin qu'il ait un contenu identique au second. Les modifications sont données sous forme de messages. Par exemple :

- **7a8** indique qu'après la septième ligne du premier fichier la ligne 8 du second doit être ajoutée (a pour *append*).
- **9d8** indique que la ligne 9 du premier fichier doit être supprimée (d pour *delete*), car elle n'existe pas derrière la ligne 8 du second.
- **5,8c4,6** indique que les lignes 5 à 8 du premier fichier doivent être remplacées (c pour *change*) par les lignes 4 à 6 du second.

Dans les trois cas de figure, les lignes précédées de < se rapportent au premier fichier et les lignes précédées de > se rapportent au second.

Des options permettent de ne pas tenir compte des espaces ou lignes vides lors de la comparaison.



L'option `-y` (ou `--side-by-side`) permet de visualiser en vis-à-vis le contenu des fichiers à comparer.



Des outils plus évolués permettent de comparer les fichiers. `wdiff` est un frontal de la commande `diff`. De même `meld` propose une interface utilisateur facilitant la comparaison.

5.3.4.1 Complément sur les expressions régulières { }

Les expressions régulières sont des suites de caractères qui permettent de réaliser des sélections qui seront utilisées par des commandes telles que `grep` ou `egrep`. Les différentes expressions régulières sont :

Expression	Description
<code>^</code>	début de ligne
<code>.</code>	un caractère quelconque
<code>\$</code>	une fin de ligne

Expression	Description
<code>x*</code>	zéro ou plusieurs occurrences du caractère « x »
<code>x+</code>	une ou plusieurs occurrences du caractère « x »
<code>x?</code>	une occurrence unique du caractère « x »
<code>[...]</code>	un intervalle de caractères autorisés
<code>[^...]</code>	un intervalle de caractères interdits
<code>\ {n\}</code>	définit le nombre de répétitions <i>n</i> du caractère placé devant

Exemples :

- « `[A-Z][A-Z]*` » : cherche les lignes contenant au minimum une majuscule (`[A-z]` : caractère permis et `[A-Z]*` : recherche d'occurrences).
- « `^[0-9]\ {3\}$` » : cherche du début à la fin les nombres (`[0-9]`) composés de 3 chiffres (`\ {3\}`).

5.3.5 Localiser des commandes

Voici trois commandes qui vous permettront de localiser les commandes et programmes sur votre système :

Commande	Description
<code>type</code>	affiche le type de la commande (alias, exécutable, etc.) et précise son chemin absolu
<code>whereis</code>	recherche une commande dans la variable <code>PATH</code> et les pages de manuel, puis retourne le ou les chemins d'accès complet(s)
<code>which</code>	renvoie le chemin absolu d'une commande

5.4 Gestion des processus

Unix est un système multitâches multi-utilisateurs, où une tâche est appelé « processus ». Un processus encapsule une commande en cours d'exécution. Plusieurs tâches (processus) peuvent être lancées en même temps :

- dans des fenêtres de terminaux différentes;
- dans une même fenêtre de terminal avec l'opérateur « `&` ».

Un processus peut être lancé :

- au premier plan (un seul à la fois par fenêtre) : il dispose du clavier et de l'écran;
- en arrière-plan (plusieurs à la fois) : il ne dispose ni du clavier ni de l'écran (tous les affichages sont perdus).

Un processus peut être :

- **actif** : en cours d'exécution;
- **stoppé** : momentanément arrêté;
- **arrêté** : définitivement arrêté.

Les commandes permettant d'interagir avec les processus sont regroupées dans le tableau suivant :

Commande	Description
<code>bg %x</code>	met en arrière-plan un programme stoppé et reprend l'exécution (% indique le numéro du processus <i>x</i> -- si % est absent, c'est le premier processus qui est considéré)
<code>fg %x</code>	passse au premier plan un programme qui était en arrière-plan
<code>jobs</code>	liste les programmes qui tournent dans un même <i>shell</i> avec leur numéro (local)
<code>kill</code>	envoie un signal à un processus (il faut en être propriétaire)
<code>killall</code>	envoie un signal à un ensemble de processus de même nom (il faut en être propriétaire)
<code>ps</code>	affiche les processus en cours d'exécution
<code>sleep</code>	suspend un processus pendant un temps donné
<code>top</code>	affiche uniquement les 10 processus qui prennent le plus de temps CPU ¹

Exemples

- **bg** : passe le programme en cours à l'arrière-plan.
- **kill -9 noproc** : arrête un processus en cours -- équivalent à « `kill -9 -1` ».
- **kill -1 noproc** : recharge le processus `init`.
- **pkill -f agent** : arrête le processus dont le nom contient « agent ».
- **killall agent** : arrête tous les processus dont le nom est « agent ».
- **kill %x** : stoppe un processus en arrière-plan.
- `Ctrl`+`c` : arrête le processus au premier plan (cf. : `kill`).
- `Ctrl`+`z` : suspend momentanément le processus au premier plan (cf. : `kill`).
- **ps -edf** (ou **-aux** sur certains systèmes) : visualise tous les numéros des processus en cours d'exécution.
- **time prog** : donne la durée de l'exécution du programme « prog ».



La séquence « commande » suivie de « `Ctrl`+`z` » et de la commande « `bg` » est similaire à « commande & ».



La commande `xkill` permet de stopper une application graphique qui ferait des siennes. Il suffit d'amener le curseur souris qui se change en croix ou en tête de mort au-dessus de la fenêtre récalcitrante, puis de cliquer dessus pour arrêter le processus.

5.5 Archivage et compression

Le tableau ci-dessous regroupe quelques commandes permettant d'archiver et de compresser des fichiers.

¹Non installées de base sur un système Unix, les commandes `htop` et `gtop` fournissent plus d'informations que la commande `top`. La commande `htop` est en principe disponible sur les machines de l'ENSICAEN.

Commande	Description
<code>compress / uncompress</code>	compression / décompression (extension « .Z »)
<code>bzip2 / bunzip2</code>	compression / décompression (extension « .bz2 »)
<code>gzip / gunzip</code>	compression / décompression (extension « .gz »)
<code>zip / unzip</code>	archivage + compression / décompression (extension « .zip »)
<code>tar</code>	archivage / désarchivage (extension « .tar »)

5.5.1 Création d'archives

La commande `tar` permet de rassembler plusieurs fichiers en un seul. Elle préserve les droits, le propriétaire et le groupe des fichiers et des dossiers, ainsi que les liens symboliques. Sa syntaxe est la suivante :

```
$ tar {-c|x|t} [options] [fichiers | dossiers]
```

La plupart du temps, une archive est ensuite compressée en utilisant une des commandes de compression `compress`, `gzip` et `bzip2` (ex. : « `tar.Z` », « `.tar.gz` » ou « `.tgz` », « `.tar.bz2` » ou « `.tbz` »).



La commande `zip` permet la création d'archives compressée. On retrouve fréquemment ce format sur les autres systèmes.

Exemples

- `tar -cvf archive.tar ./tp06` : archive le sous-dossier « `tp6` » présent dans le dossier courant;
- `tar -xvf archive.tar` : extrait l'archive dans le dossier courant;
- `tar -tvf archive.tar` : visualise le contenu de l'archive;
- `tar -czvf archive.tar.gz ./tp06` : crée une archive compressée (avec `gzip`) du sous-dossier « `tp06` »;
- `tar -czvf archive.tar.gz ./tp06` : crée une archive compressée (avec `gzip`) du sous-dossier « `tp06` »;
- `tar -cv tp06 | gzip > archive.tar.gz` : même chose;
- `tar -xzvf archive.tar.gz` : extrait dans le dossier courant l'archive compressée avec `gzip` lors de l'archivage;
- `tar -xjvf archive.tar.bz2` : extrait dans le dossier courant l'archive compressée avec `bunzip2` lors de l'archivage;
- `zip -r archive.zip ./tp06` : crée une archive compressée du sous-dossier « `tp06` »;
- `unzip archive.zip` : extrait dans le dossier courant l'archive compressée avec `zip`.

5.6 Réseau

5.6.1 Connexion et transfert

Voici quelques commandes qui vous permettront de travailler à distance en exécutant des commandes Unix sur le serveur Cybèle de l'ENSICAEN ou encore en transférant des fichiers entre différentes machines.

Commande	Description
<code>curl</code>	transfert des données de ou vers un serveur (supporte de nombreux protocoles)
<code>scp</code>	copie distante sécurisée ²
<code>sftp</code>	client FTP en mode sécurisé (SFTP ³)
<code>ssh</code>	lance un client SSH ⁴
<code>wget</code>	télécharge des fichiers via les protocoles HTTP, HTTPS et FTP



Si les environnements en local et à distance ont des serveurs graphiques compatibles, alors il est possible de lancer `ssh` avec l'option `-X`.

Exemples

- `curl https://www.ecole.ensicaen.fr` : récupère et affiche le code HTML ⁵ transmis par le serveur.
- `curl http://cdim...lubuntu-20.04.iso --output lubuntu.iso` : télécharge le fichier « `lubuntu-20.04.iso` » du serveur et le copie localement dans « `lubuntu.iso` ».
- `ssh jsaigne@cybele.ecole.ensicaen.fr` : connecte l'utilisateur « `jsaigne` » (Jean Saigne) au serveur Cybèle en utilisant SSH.
- `scp ./im.bmp jsaigne@cybele.ecole.ensicaen.fr:~/Documents` : copie le fichier local « `im.bmp` » vers le dossier « `~/Documents` » sur Cybèle.
- `scp -r ./tp1 jsaigne@cybele.ecole.ensicaen.fr:~/od1` : copie du sous-dossier local « `tp1` » vers le dossier distant « `~/od1` » sur Cybèle.
- `scp jsaigne@cybele.ecole.ensicaen.fr:~/od1/tp1/LISEZMOI.md ./` : télécharge localement dans le dossier courant le fichier « `LISEZMOI.md` ».
- `wget www.ecole.ensicaen.fr/~jsaigne/adresse.html` : télécharge dans le dossier courant local le fichier distant « `adresse.html` ».



Le logiciel `httptrack` permet d'aspirer des sites. Il résulte d'un travail fait en projet à l'ENSICAEN en 1999 par Roche et Philippot.

5.6.2 Communication

Commande	Description
<code>finger</code>	donne les caractéristiques d'un utilisateur
<code>groups</code>	donne la liste des groupes auxquels appartient un utilisateur
<code>users</code>	liste des utilisateurs connectés

²SCP (*secure copy*) est un protocole de copie utilisant SSH.

³SFTP (*secure file transfert protocol*) est un protocole de transfert de fichiers utilisant SSH.

⁴SSH (*secure shell*) est un protocole de connexion avec clé de chiffrement.

⁵HTML (*hypertext markup language*) est le langage balisé utilisé sur le Web (cf. cours).

Commande	Description
who (rwho)	liste des utilisateurs connectés sur toutes les machines du domaine
whoami	retourne l'identifiant de l'utilisateur courant
talk	permet de discuter avec un autre utilisateur

Exemples

- **finger jsaigne** : donne les caractéristiques de l'utilisateur.
- **groups jsaigne** : donne la liste des groupes auxquels appartient l'utilisateur.
- **talk jsaigne** : permet de discuter avec un l'utilisateur.

5.7 Commandes diverses

Le tableau suivant comporte un ensemble de commandes qui pourront vous être utiles, notamment la commande `date` :

Commande	Description
date	affiche la date courante
ls -l	affiche la liste des fichiers ouverts sur le système (vous risquez d'être surpris!)
uname -a	affiche des informations sur la version du noyau et sur le type d'architecture du processeur

5.8 Complément sur la protection des fichiers

Commande `umask`

Afin d'attribuer des droits d'accès par défaut à un dossier ou fichier, il faut lancer la commande `umask` avec une valeur particulière. Cette valeur est obtenue en calculant la différence entre 7 et chaque chiffre composant la valeur des droits d'accès. Ainsi, pour attribuer par défaut la valeur 754 (tous les droits pour l'utilisateur, droits de lecture et d'exécution pour le groupe et droit de lecture pour les autres), il faut fournir à la commande `umask` la valeur 023 ($7-7=0$, $7-5=2$ et $7-4=3$) :

```
$ umask 023
```

Cette ligne pourra figurer dans le fichier d'initialisation `.profile` afin que chaque session, ces droits soient pris en compte.

Droits d'accès étendus

Pour définir des droits d'accès étendus qui permettent à un utilisateur d'effectuer une opération sur des fichiers lui appartenant, comme modifier son mot de passe par exemple), il faut utiliser l'option `+s` ou rajouter les valeurs 4000 pour l'utilisateur et 2000 pour le groupe.

Pour l'utilisateur : « `chmod 4755` » et pour le groupe : « `chmod 2755` »

Droits d'accès et /tmp

Tout utilisateur peut copier des fichiers dans le dossier /tmp. Afin de garantir que ces fichiers ne puissent pas être effacés par un autre utilisateur que celui qui les a copiés (hormis le superutilisateur), il faut positionner le *sticky bit* de la manière suivante :

```
$ sudo chmod u+t /tmp
```

ou encore :

```
$ sudo chmod 1777 /tmp
```

5.9 tmux : un multiplexeur de terminal

Dans un monde où les interfaces graphiques sont la norme, les terminaux peuvent apparaître comme trop exigus. C'est la raison pour laquelle on a vu apparaître des *multiplexeurs* de terminal tels que GNU Screen ou *tmux* que nous utilisons ici (*tmux* et sa documentation sont disponibles à l'adresse : <https://github.com/tmux/tmux/wiki>).

Le multiplexeur *tmux* fonctionne en mode « client-serveur » et manipule trois types d'éléments comme le montre la figure 5.3 : des **sessions**, des **fenêtres** et des **panneaux**.

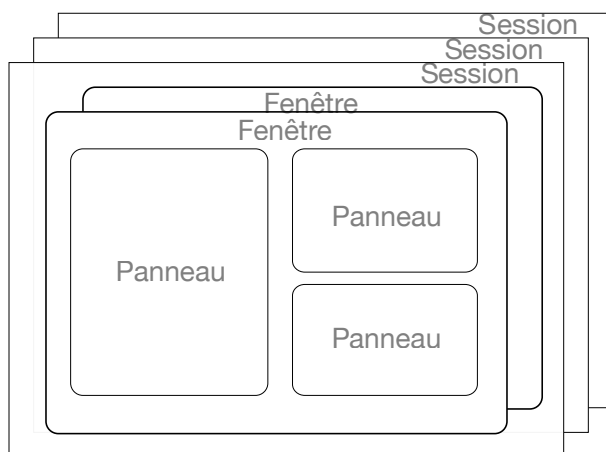


Fig. 5.3 : Structure de *tmux*.

- Une **session** est associée à un « client » et elle peut regrouper plusieurs terminaux « virtuels » (locaux ou distants) qui y sont « attachés ». Une session peut temporairement être « détachée » du serveur *tmux* sans pour autant continuer de fonctionner en arrière-plan, puis rattachée si nécessaire.
- Une **fenêtre** recouvre le terminal (et donc la session) depuis lequel a été lancé *tmux*. Si une session peut avoir plusieurs fenêtres, elle ne peut en afficher qu'une seule à la fois. Une fenêtre peut être divisée en plusieurs panneaux.
- Un **panneau** correspond à une partie de la fenêtre et contient un terminal virtuel.

Par exemple, on pourrait créer une session pour les projets en cours, des fenêtres pour chaque projet et découper chaque fenêtre en un panneau pour l'éditeur et un panneau pour le test de l'application. La figure 5.4 montre une session *tmux* composée d'une fenêtre découpée en trois panneaux dans lesquels s'exécutent *Vim*, un interpréteur *bash* et la commande *htop*.

```

14 [~/]: # (-----)
15
16 # Compilation
17
18 > "_#define QUESTION ((bb) || ((bb))" _" _Shakespeare_
19
20 [~/]: # (-----)
21
22 Dans ce chapitre, nous passons en revue les différentes étapes de la
23 compilation d'un programme écrit en langage C. Nous utilisons l'outil GCC
24 (.GNU Compiler Collection.) qui intègre un certain nombre de compilateurs
25 associés à des langages tels que le C, le C++, Objective-C, le Go, etc.
26 Pour plus d'informations, le lecteur curieux pourra se reporter à la
27 documentation officielle @Stallman2003.
28
29 [~/]: # (-----)
30
31 # Étapes de compilation
32
33 La **compilation** regroupe plusieurs étapes, toutes réalisées à l'aide
34 de la commande 'gcc' :
35
36 - **précompilation** ;
37 - **traduction** ;
38 - **optimisation** ;
39 - **assemblage** ;
40 - **édition de liens** .
41
42 Ces différentes étapes nécessitent des fichiers en entrées et produisent des
43 fichiers de sortie comme l'indique la figure -#fig:compilation ainsi que le
44 tableau suivant.
45
46 | Étape | Fichier en entrée | Fichier en sortie |
47 |-----|-----|-----|
48 | Précompilation | .c | .c |
49 | Traduction | .c | .o |
50 | Optimisation | .c | .o |
51 | Assemblage | .o | .o |
52 | Édition de liens | .o | exécutable |
53
54 [Chaine de compilation permettant de produire l'exécutable "bonjour".](figures/compilation/compilation.pdf){#fig:compilation}
55
56 GNU gcc, est un "frontal" (.front-end, en anglais) qui réalise les
57 différentes étapes de la compilation en appelant les outils adéquats en
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

```

Fig. 5.4 : Exemple de composition avec tmux.

Avant d'aller plus loin, voici comment lire les commandes et raccourcis clavier que nous allons rencontrer :

- **C-b%** signifie que l'on doit appuyer sur la touche **Ctrl** et la touche **b**, qu'on les relâche, puis que l'on appuie sur la touche **%**.
- **q** signifie que l'on doit appuyer sur la touche **q**.

Gestion des sessions

Commande	Description
tmux	Lance <i>tmux</i> et ouvre une nouvelle session
C-b ?	Lance l'aide en ligne de <i>tmux</i> (Q pour quitter)
tmux ls	Liste les différentes sessions
C-b d	Détache la session active
tmux a	Attache la dernière session active
tmux new -s test	Crée et attache la session de nom « test »
C-b \$	Renomme la session active
C-b s	Passé d'une session à une autre
C-b)	Passé à la session suivante
C-b (Passé à la session précédente
tmux kill-session -t test	Détruit la session « test »
tmux kill-server	Détruit toutes les sessions

Gestion des fenêtres

Commande	Description
C-b c	Crée une nouvelle fenêtre
C-b x	Détruit la fenêtre courante
C-b n	Passe à la fenêtre suivante
C-b p	Passe à la fenêtre précédente
C-b ,	Renomme la fenêtre active
C-b &	Ferme la fenêtre active

Gestion des panneaux

Commande	Description
C-b %	Découpage vertical
C-b "	Découpage horizontal
C-b + flèches	Passe d'un panneau à un autre
exit	Ferme le panneau actif

Exercices

Exercice 5.1

À l'aide d'une seule commande, créez l'arborescence de dossiers vides « d1/sd2/ssd3/sssd4/ » dans votre dossier personnel. Sans vous déplacer, copiez dans chacun de ces dossiers le fichier « loremipsum.txt » dans lequel trouverez dans le dossier « /home/public/tp_od1/tp01/ ». Quels sont les droits de ce fichier ? Modifiez les droits d'accès du fichier que vous venez de copier en permettant une lecture par tous les utilisateurs et une écriture par vous uniquement. Supprimez l'ensemble des fichiers et dossiers créés.

Exercice 5.2

Comment supprimer le fichier « -r! » ?

Exercice 5.3

Depuis le terminal, exécutez les commandes suivantes :

```
$ vim io.c & top &
$ jobs
$ fg %1
$ C-z
$ bg %1
$ kill -9 %1
$ fg %2
```

Exercice 5.4

À l'aide du multiplexeur *tmux*, créez une session nommée « projetQuizz » et comprenant 5 panneaux disposés de la manière suivante :

```
+-----+-----+
|               |               |
|               |               |
|               |               |
+-----+-----+-----+
|               |               |               |
|               |               |               |
|               |               |               |
+-----+-----+-----+
```

Exercice 5.5

Quels sont les droits par défaut donnés à chaque fichier dans les cas suivants :

- `umask 022`
- `umask 077`
- `umask 026`

6 Vim : un éditeur évolué pour développeur

« How do you generate a random string?... » « Put a web designer in front of VIM and tell him to save and exit. » lol.browserling.com.

Dans ce chapitre, nous présentons un outil fondamental pour tout développeur : l'éditeur.

6.1 Introduction

Les développeurs disposent d'un certain nombre d'outils pour éditer du code et le remettre en forme d'un simple clic de souris ou à l'aide d'un raccourci clavier si nécessaire. Le choix de l'outil dépend du langage utilisé, du système d'exploitation sur lequel l'application doit être réalisée ainsi que du budget disponible si l'outil est payant. Certains outils sont monolangage et ne sont disponibles que sur une plateforme donnée, alors que d'autres sont multilingages et multiplateformes. La figure 6.1 présente les environnements de développement les plus utilisés par les développeurs d'applications en 2021.

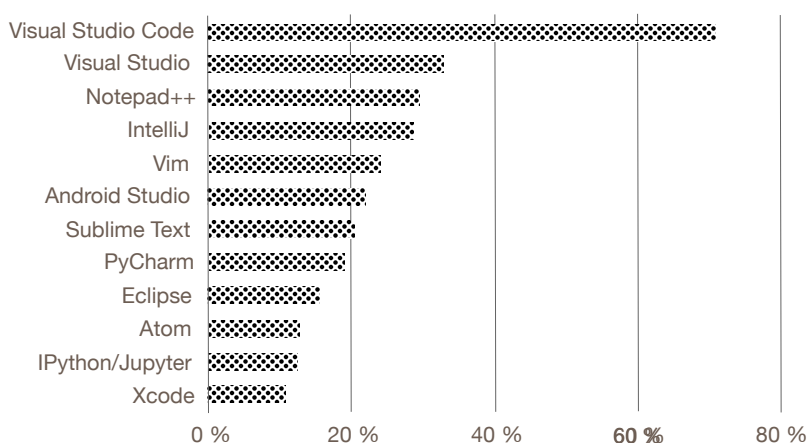


Fig. 6.1 : Environnements populaires chez les développeurs en 2021 (source : stackoverflow.com).

Parmi les outils de développement, en voici quelques-uns qui s'exécutent dans le terminal et permettent une utilisation sur de vieilles machines, voire à distance :

Éditeur	Fonctionnalités	Courbe d'apprentissage	Site officiel
<i>Vim</i>	+++++	longue	https://www.vim.org
<i>Emacs</i>	+++++	longue	https://www.gnu.org/software/emacs/
<i>Micro</i>	+++	rapide	https://micro-editor.github.io

Éditeur	Fonctionnalités	Courbe d'apprentissage	Site officiel
<i>nano</i>	++	rapide	https://www.nano-editor.org

Dans ce cours, nous avons décidé d'utiliser l'éditeur *Vim* qui est une version améliorée de l'éditeur Unix de base *vi*, et qui présente un grand nombre de fonctionnalités que nous détaillons plus loin.

6.2 Intérêt pour le développement logiciel

Voici quelques points qui justifient l'utilisation d'un outil tel que *Vim* :

- Il est multi-usage et diffère des éditeurs dédiés tels que Visual Studio, CLion ou Netbeans.
- Il est puissant en matière de fonctionnalités pour la programmation.
- Il est configurable et extensible.
- Il se pilote avec des commandes clavier, mais le support souris peut être activé.
- Il est libre et *open source*.
- Portable sur tous les systèmes d'exploitation (Ms-Windows, Mac OS X, Linux).
- Utilisé par de nombreux développeurs.

6.3 Lancement de l'éditeur

L'éditeur *Vim* peut être lancé de deux manières depuis le terminal :

Commande	Description
<code>vim [options]</code>	Lance <i>Vim</i> sans ouvrir de fichier
<code>vim [options] [fichiers..]</code>	Ouvre <i>Vim</i> avec les fichiers spécifiés

6.4 Modes principaux de Vim

L'éditeur *Vim* possède trois modes principaux :

- **Mode interactif** (ou « normal ») : c'est le mode par défaut en lançant l'éditeur. Il permet de se déplacer dans le texte, de supprimer des lignes, de copier-coller du texte, etc. À partir du mode *insertion*, on y revient en appuyant sur la touche `[Esc]`.
- **Mode insertion** : c'est le mode permettant l'éditoin de texte. Pour y accéder il faut depuis le mode interactif, appuyer sur `[i]` (*insert*) et pour en sortir il faut appuyer sur la touche `[Esc]`.
- **Mode commande** : ce mode permet de lancer des commandes telles que « quitter » (`[q]`), « enregistrer » (`[w]`), etc. Il est aussi utilisé pour activer des options comme la coloration syntaxique, l'affichage des numéros de ligne, etc. Pour activer ce mode, il faut être en mode interactif et appuyer sur la touche `[:]`. La commande doit alors être validée avec la touche `[Enter]` afin de revenir au mode interactif.

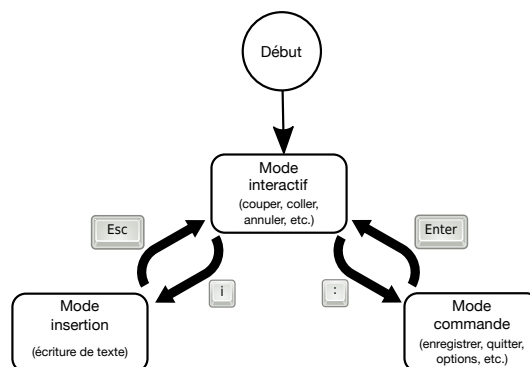


Fig. 6.2 : Modes principaux de Vim.

Avant d'aller plus loin, voici comment lire les commandes et raccourcis clavier que nous allons rencontrer :

- C-u signifie que l'on doit appuyer sur la touche `Ctrl` et la touche `u`.
- 2dd est une commande du mode interactif qui signifie que l'on doit appuyer sur la touche `2` que l'on relâche, puis sur la touche `d` deux fois de suite (il ne faut surtout pas appuyer sur `Enter`).
- :q signifie que l'on doit d'abord appuyer sur la touche `:` afin de passer dans le mode « commande », puis on la relâche, et on appuie sur la touche `q` suivie d'un appui sur la touche `Enter` afin de valider la commande.

6.5 Commandes de Vim

La syntaxe des commandes *Vim* peut se résumer à l'utilisation de *verbes* et de combinaisons de *verbes* et d'*objets*.

Par exemple, le verbe « delete » (supprimer) est associé au caractère `d` et l'objet « word » (mot) au caractère `w`. Ainsi la commande `dw` signifie « supprimer le mot » sur lequel se trouve le curseur. On trouvera entre autres les verbes : « **y**ank » (copier), « **p**aste » (coller), « **i**nsert » (insérer), « **q**uit » (quitter), « **w**rite » (enregistrer), etc.

Les commandes peuvent être répétées :

- « `.` » permet de répéter la dernière commande entrée;
- « `5d` » permet de répéter 5 fois la commande `d`.

et annulées :

- « `u` » (*undo*) : permet d'annuler la dernière commande entrée;
- « `5u` » : permet d'annuler les 5 dernières annulations;



Apprendre les commandes clavier au fur et à mesure vous permettra d'atteindre une rapidité d'exécution, ainsi qu'une indépendance relative aux interfaces graphiques, par exemple pour le cas où vous auriez à éditer du code sur une machine distante qui n'est accessible qu'au travers d'un tunnel **ssh** comme c'est le cas avec le serveur Cybèle.

6.6 Fenêtre principale de Vim

Contrairement à l'éditeur *Emacs*, *Vim* ne comporte pas de menu dans sa version de base, mais simplement une zone principale pour l'édition, ainsi qu'une barre d'état ou invite de commandes placée en bas comme le montre

la figure 6.3.

6.7 Mode insertion

Depuis le mode interactif, on passe dans le mode « insertion » en appuyant sur la touche `i` (pas besoin d'appuyer sur `Enter`). L'indication « -- INSERTION -- » apparaît alors dans la barre d'état comme le montre la figure 6.3.

```

2 #include <stdlib.h>
3
4 /**
5  * Sort an array in ascending order.
6  * @param array The array to be sorted.
7  * @param length The length of the array.
8  */
9 void sort(int *array, int length) {
10     for (int i = 0; i < length - 1; i++) {
11         for (int j = 0; j < length - i - 1; j++) {
12             if (array[j] > array[j + 1]) {
13                 int temp = array[j];
14                 array[j] = array[j + 1];
15                 array[j + 1] = temp;
16             }
17         }
18     }
19 }
20
21 int main(void) {
22     int array[5] = {5, 3, 2, 4, 1};
23
24     for (int i = 0; i < 5; i++) {
25         printf("%d ", array[i]);
26     }
27
28     sort(array, 5);
29
30     for (int i = 0; i < 5; i++) {
31         printf("%d ", array[i]);
32     }
33
34     return 0;
35 }

```

-- INSERTION -- 3,1 Bas

Fig. 6.3 : Fenêtre de l'éditeur Vim en mode insertion.

Le tableau suivant présente les commandes qui permettent d'insérer du texte, mais aussi de passer en mode « insertion ».

Commande	Verbe	Description
<code>i</code>	<i>insert</i>	insérer avant le curseur
<code>I</code>	<i>Insert</i>	insérer en début de ligne
<code>a</code>	<i>append</i>	ajouter du texte après le curseur
<code>A</code>	<i>Append</i>	ajouter du texte en fin de ligne

6.8 Déplacement du curseur

En mode interactif, les touches `h` (gauche), `j` (bas), `k` (haut) et `l` (droite) permettent de déplacer le curseur dans l'éditeur. Les flèches du clavier, si elles sont disponibles, permettent aussi de déplacer le curseur.

En mode interactif, on peut aussi utiliser d'autres commandes pour se déplacer rapidement. Entre autres :

- `0` (origin) : permet de se placer en début de ligne;

- **\$** : permet de se placer en fin de ligne;
- **w** : permet de se déplacer de mot en mot;
- **8j** : permet de se déplacer de 8 lignes vers le bas;
- etc.

Toujours en mode interactif, on peut aussi réaliser des sauts dans le texte, par exemple à l'aide des commandes **G** (**Go to**) ou **g** :

- **8G** : saute à la ligne 8;
- **G** : saute à la dernière ligne;
- **gg** : saute à la première ligne.

6.9 Opérations standards

Le tableau suivant regroupe un certain nombre de commandes de base :

Commande	Verbe	Description
:w	<i>write</i>	écrire (enregistrer) le fichier
:q	<i>quit</i>	quitter (échoue si modifications non sauvegardées)
d	<i>delete</i>	supprimer (couper) des mots, des lignes
y	<i>yank / copy</i>	copier des mots, des lignes
p / P	<i>paste</i>	coller le presse-papier après le curseur / avant le curseur
r / R	<i>replace</i>	remplacer un caractère / mode remplacement
u	<i>undo</i>	annuler la dernière commande

6.9.1 Enregistrer

Si *Vim* a été lancé sans qu'un nom de fichier n'ait été indiqué, il faut utiliser la commande « **:w nomfichier** » pour enregistrer le contenu courant. Afin d'enregistrer le fichier en cours d'édition, on utilise « **:w** ».

6.9.2 Quitter

Afin de quitter l'éditeur, on utilise la commande « **:q** » (« **:q!** » pour forcer la sortie). De plus, si l'on souhaite enregistrer le fichier avant de quitter on peut combiner les commandes : « **:wq** ».



Lorsque vous travaillez, si vous avez des modifications non enregistrées, *Vim* crée et enregistre périodiquement une sauvegarde des tampons ouverts dans des fichiers dont le nom est du type « **.nom_fichier.swp** ». Dès que vous enregistrez vos tampons et que vous quittez l'éditeur, les fichiers sont supprimés du dossier courant. Toutefois, si vous quittez *Vim* sans enregistrer vos modifications ou si, pour une raison quelconque, *Vim* se ferme anormalement, les fichiers seront conservés et pourront être utilisés pour récupérer vos modifications.

Pour récupérer les modifications lorsqu'un fichier a été fermé anormalement, lancez *Vim* avec l'option **-r** (*recover*) suivie du nom du fichier d'échange.

6.9.3 Couper

Les commandes `d` (*delete*) ou son raccourci `x` (*excise*¹) permettent de réaliser des coupures dans le texte. Le texte coupé est alors mémorisé dans le presse-papier de *Vim* (mais pas celui de Gnome!). Voici quelques exemples :

- **dw** : efface le mot placé sous le curseur (si le curseur est en début de mot) ou la fin du mot si le curseur est placé au milieu du mot;
- **3dw** : efface trois mots à partir du mot placé sous le curseur;
- **dd** : efface toute la ligne;
- **5dd** : efface les cinq lignes à partir de la ligne courante;
- **d0** : efface tout caractère entre le début de ligne et le caractère sous le curseur;
- **d\$** : efface tout caractère entre le caractère sous le curseur et la fin de ligne;
- **x** : efface le caractère placé sous le curseur;
- **7x** : efface les sept caractères à partir du caractère placé sous le curseur.

6.9.4 Copier

La copie d'un texte s'effectue à l'aide de la commande `y` pour *yank* (tirer, arracher en français). Voici quelques exemples :

- **yw** : copie le mot placé sous le curseur (si le curseur est en début de mot) ou la fin du mot si le curseur est placé au milieu du mot;
- **3yw** : copie trois mots à partir du mot placé sous le curseur;
- **yy** : copie la ligne en mémoire;
- etc.

6.9.5 Coller

Le collage d'un texte s'effectue à l'aide de la commande `p` pour *paste*. Tout texte coupé ou copié avec les commandes précédentes sera collé sur la ligne après le curseur.

- **p** : colle le texte préalablement coupé;
- **7p** : colle sept fois de suite le texte préalablement coupé.

6.9.6 Remplacer

Il est possible de remplacer un ou plusieurs caractères consécutifs à l'aide de la commande `r` (*replace*). Par exemple :

- **rL** : remplace le caractère placé sous le curseur par la majuscule L;
- **4rL** : remplace les quatre caractères à partir du caractère placé sous le curseur par la majuscule L.



La commande `R` place l'éditeur en mode « remplacement » et permet de remplacer plusieurs caractères à la fois. Pour revenir au mode interactif, appuyez sur `[Esc]`.

¹Le verbe *excise* (exciser en français) est une commande de suppression de texte fréquemment rencontrée sous Unix. La commande fut initialement utilisée par l'éditeur `ed` pour des suppressions de lignes.

6.9.7 Annuler

Toute action non souhaitée peut être annulée à l'aide de la commande `u` (*undo*) :

- `u` : annule la dernière modification ;
- `3u` : annule les trois dernières modifications.

6.9.8 Passer une lettre (ou plusieurs lettres) en majuscule(s)

La commande « `~` » suivi d'un mouvement du curseur (flèche droite par exemple), permet de transformer la lettre minuscule sous le curseur en majuscule ou *vice versa*. C'est très pratique pour obtenir une majuscule accentuée. *Vim* fait encore plus :

- `3~` : transformation des 3 caractères suivants ;
- `g~$` : même chose pour le mot sous le curseur ;
- `g~3w` : même chose pour les 3 mots suivants ;
- `g~~` : même chose pour la ligne entière.

6.10 Opérations avancées

Le tableau suivant regroupe un certain nombre de commandes avancées :

Commande	Verbe	Description
<code>:bn</code>	<i>buffer next</i>	passage au tampon suivant
<code>:bp</code>	<i>buffer previous</i>	passage au tampon précédent
<code>:bd</code>	<i>buffer delete</i>	fermeture du tampon courant
<code>:ls</code>	<i>list</i>	retourne la liste des tampons ouverts
<code>:sp</code>	<i>split</i>	découpage horizontal de la fenêtre
<code>:vsp</code>	<i>vertical split</i>	découpage vertical de la fenêtre
<code>:s</code>	<i>search and replace</i>	recherche et remplacement de texte
<code>:r</code>	<i>merge</i>	fusion de fichiers

6.10.1 Tampons

Vim utilise un mécanisme de tampons (*buffers*) afin d'éditer plusieurs fichiers à la fois. En règle générale, il est préférable d'utiliser plusieurs tampons plutôt que d'exécuter plusieurs instances de *Vim* en même temps. Cela évite, par exemple, la modification d'un même fichier depuis deux instances de *Vim* différentes.

L'ouverture de plusieurs fichiers peut se faire au lancement de *Vim* :

```
$ vim fic1.c fic1.h fic2.c fic2.h main.c README.md
```

Le tampon courant (celui qui est visible) est alors associé au fichier « `fic1.c` ».

Pour passer d'un tampon à un autre, on utilisera les commandes « `:bn` » ou « `:bp` », et pour fermer le tampon courant, on entrera la commande « `:bd` ».



Si on a lancé *Vim* sans argument et si l'on souhaite ouvrir un fichier depuis la fenêtre courante (ou encore depuis l'onglet courant), on peut utiliser la commande « :e fichier » (ou encore « :n fichier »).

6.10.2 Multifenêtrage

L'affichage de plusieurs tampons simultanément peut être réalisé par des commandes qui permettent de scinder la fenêtre : « :sp » (séparation horizontale) et « :vsp » (séparation verticale). Chaque sous-fenêtre (correspondant à un tampon) est appelée *viewport* que nous traduirons par « panneau ».

La commande « :sp fic.txt » ouvre le fichier « fic.txt » dans un deuxième panneau placé au-dessous de celui du tampon courant, alors que la commande « :sp » seule rouvre le tampon courant dans ce deuxième panneau.

Voici une liste de raccourcis à utiliser lorsque la fenêtre de *Vim* est scindée en plusieurs panneaux :

Commande	Description
C-w C-w	navigue entre les panneaux
C-w h (ou j, k, l)	se déplace vers le panneau gauche
C-w +	agrandit le panneau courant
C-w -	réduit la taille du panneau courant
C-w =	réinitialise la taille des panneaux
C-w r	échange la position des panneaux
C-w q	ferme le panneau courant



Si l'on souhaite ouvrir plusieurs fichiers en mode multifenêtre, il est possible de lancer *Vim* avec l'option -O :

```
$ vim -O fic1 fic2 ...
```

Lorsque deux tampons sont ouverts, il est possible de les afficher dans deux fenêtres verticales côte à côte à l'aide de la commande « :vert ».

6.10.3 Onglets

Pour ceux qui ne souhaitent pas visualiser l'ensemble des tampons en mode multifenêtrage, *Vim* dispose aussi d'un mécanisme d'onglets (qui supportent le multifenêtrage).

Commande	Description
:tabnew	ouvre un nouvel onglet
:tabnew fichier	ouvre fichier dans un nouvel onglet
:tabn	affiche l'onglet suivant

Commande	Description
:tabp	affiche l'onglet précédent
:tabc	ferme l'onglet courant

6.10.4 Rechercher et remplacer

Pour une recherche seule, la commande « / » entrée depuis le mode interactif permet de passer en mode « recherche ». Le curseur se place alors en bas de l'éditeur et affiche « / ». Il suffit d'indiquer le mot recherché et d'appuyer sur `Enter`. Les différentes occurrences du mot sont soulignées et le curseur se place alors sur la première occurrence rencontrée. Pour passer aux occurrences suivantes, il suffit d'appuyer sur « n » (ou « N » pour les occurrences précédentes).

Afin de rechercher et remplacer du texte dans un tampon, on utilise la commande « :s » ou « :%s ». Ainsi, pour remplacer le mot « ancien » par le mot « nouveau », on mettra en oeuvre une des variantes suivantes :

Commande	Description
:s/ancien/nouveau	remplace la 1re occurrence de la ligne du curseur
:s/ancien/nouveau/g	remplace les occurrences de la ligne du curseur
:8,22s/ancien/nouveau/g	remplace les occurrences des lignes 8 à 22
:%s/ancien/nouveau/g	remplace les occurrences du tampon
:%s/ancien/nouveau/gc	<i>idem</i> avec confirmation
:%s/ancien/nouveau/gci	<i>idem</i> sans considération minuscules/majuscules.

Voir ce [site](#) pour plus de détails.

6.10.5 Fusionner des fichiers

La commande « :r fic » permet d'insérer le fichier « fic » à la position du curseur dans le tampon courant.

6.10.6 Comparer des fichiers

Nous avons déjà rencontré la commande Unix `diff` qui permet de mettre en évidence les différences entre deux fichiers. *Vim* possède un mode permettant de réaliser cette analyse. Par exemple, pour comparer les fichiers « `fic1` » et « `fic2` » qui sont ouverts dans deux fenêtres, on exécute la commande « :diffthis » dans chacune des deux fenêtres afin de visualiser les différences.

6.10.7 Lancement de commandes externes

Il est possible d'exécuter des commandes du *shell* directement depuis *Vim*. Il suffit d'entrer le caractère « ! » suivi par le nom de la commande. Par exemple : « :!ls ».

6.10.8 Indenter du code

En mode « insertion », le code peut être indenté à l'aide de tabulations. En mode interactif, on indente la ligne sur laquelle se trouve le curseur en entrant deux fois le caractère « > » (>>) (désindentation avec <<).

Il est aussi possible d'indenter proprement la totalité du code en utilisant la commande « gg=G » : gg pour aller au début du tampon, = permet de corriger l'indentation et G permet de réaliser l'opération jusqu'à la fin du fichier.



Indenter un code ne suffit pas pour le mettre totalement en forme en respectant les règles vues dans ce cours. Nous renvoyons le lecteur à l'excellent outil *Artistic Style* que nous présentons dans le chapitre 8.

6.10.9 Chiffrer vos fichiers texte

Vim dispose d'un moyen de chiffrer vos fichiers texte à l'aide de l'algorithme [Blowfish](#).

Pour créer un nouveau document chiffré, il suffit d'utiliser l'option « -x » lors de l'appel à Vim :

```
$ vim -x journal_intime.md
```

L'application demande alors d'entrer un mot de passe et l'édition peut commencer.

Dans le cas où le fichier est déjà en cours d'édition et que l'on souhaite le chiffrer, il suffit d'entrer la commande « :X » (attention c'est une majuscule cette fois-ci), puis d'entrer le mot de passe à l'invite.

6.11 Configuration de Vim

Vim peut être personnalisé de deux façons différentes :

- en activant ou désactivant des options de Vim (voir <http://vimdoc.sourceforge.net/html/doc/>);
- en installant des greffons (voir <https://www.vim.org/scripts/>).

6.11.1 Activation des options en mode commande

Les options peuvent être activées à l'aide de la commande set après le démarrage de Vim, mais elles ne seront prises en compte que pendant la durée d'utilisation de Vim :

- **:set nom_option** : activer une option;
- **:set nonom_option** : désactiver une option en ajoutant le préfixe no à son nom;
- **:set nom_option=valeur** : pour les options qui prennent une valeur;
- **:set nom_option?** : connaître l'état d'une option.

6.11.2 Fichier de configuration de Vim

Le fichier de configuration « .vimrc » (à placer dans le dossier personnel) permet d'activer des options et greffons de manière permanente. Voici un exemple de fichier de configuration ² :

²Le site <https://vimconfig.com> permet de générer un fichier « .vimrc » personnalisé.

```
" Annule la compatibilité avec l'ancêtre Vi : totalement indispensable
set nocompatible

" -- Affichage --

set title      " Met à jour le titre de la fenêtre ou du terminal
set number     " Affichage des numéros de ligne
set ruler      " Affichage la position actuelle du curseur
set wrap       " Affichage des lignes trop longues sur plusieurs lignes
set textwidth=80 " Largeur limitée à 80 caractères
set showcmd    " Affichage de la commande en cours
set hidden     " Cache les fichiers lors de l'ouverture d'autres fichiers
set autoindent " Indentation automatique
set cindent    " Indentation C

" -- Clavier et souris --
set backspace=indent,eol,start " Active le comportement 'habituel' de
                                " la touche retour en arrière
set mouse=a                  " Activation du support souris

" -- Recherche --
set ignorecase " Ignorer la casse lors de la recherche
set smartcase  " Si une recherche contient une majuscule, re-active
                " la sensibilité à la casse
set incsearch  " Surligne les résultats de recherche pendant la saisie
set hlsearch   " Surligne les résultats de recherche

" -- Sons --
set visualbell " Empêche Vim de beeper
set noerrorbells " Empêche Vim de beeper

" -- Thème, couleurs, coloration syntaxique --
syntax on      " Activation de la coloration syntaxique
set background=dark " Fond sombre
let g:gruvbox_contrast_dark = 'hard' " Contrast fort
colorscheme gruvbox " Thème de couleurs Gruvbox

" -- Divers --
set antialias
autocmd BufNewFile,BufRead *.c set formatprg=astyle\ -A14Hpj
autocmd BufNewFile,BufFilePre,BufRead *.md set filetype=markdown.pandoc
```

6.12 Conclusion

Quel que soit l'éditeur dédié à la programmation, il faut se l'approprier et le connaître parfaitement. *Vim* possède un grand nombre de greffons (*plugins*) que vous pourrez trouver en consultant vimawesome.com/ ou en mettant en action votre moteur de recherche. Pour des informations plus complètes sur *Vim*, vous pouvez vous reporter par exemple à l'ouvrage d'A. Robbins [10]. Si vous préférez utiliser un concurrent de *Vim*, tel qu'*Emacs*, consultez le livre de D. Cameron [4].

Exercices

Exercise 6.1

Lancez *Vim* et collez-y le texte ci-dessous, puis tentez de suivre la ligne en utilisant les touches **h**, **j**, **k**, et **l**. Si vous commencez accidentellement à taper du texte, appuyez sur **Esc** pour quitter le mode insertion et sur **u** pour annuler les modifications.

V

Bien joué !

Exercise 6.2

Lancez *Vim* et collez-y le texte ci-dessous que vous corrigerez en supprimant les caractères inutiles (commande de suppression `x`).

C'esst aiiqqnsi qu'un soavir d'hieever, Arsèneee LLuupiin me raacontqta l'hisutouire de sons arresetation. Le hasard d'incipdents donte j'écriraies queleque jourtt le réucit avaientt nouxxé entre nous desse lienssss... dirraiz-je d'amittié ? Oui, j'ose croire qu'Arsèneee LLuupiin m'honore de quelquelque amittié, et que c'est par amittié qu'il arrive parfois chez mouaoi à l'improviste, apportant, dans le silence de mon cabbbinet de travail, sa gaieté juvéénile, le rayonnement de sa vie ardente, sa beillle humoeur d'hompmme pour qui la destinée n'a que faveurs et sourires.

Exercise 6.3

Lancez *Vim* et collez-y le texte ci-dessous, puis faites correspondre les paires de lignes de texte en insérant le texte manquant (commande d'insertion *i*).

Il m'évisagea prodément, puis il me dies yeux dans les yeux :
Il me dévisagea profondément, puis il me dit, les yeux dans les yeux :

- Arpin, n'estpas ?
- Arsène Lupin, n'est-ce pas ?

Je ms à rie.

Je me mis à rire.

- Nonard d'Andrézout simment.
- Non, Bernard d'Andrézy, tout simplement.
- Bernézy est mort il y a trans en Madoine.
- Bernard d'Andrézy est mort il y a trois ans en Macédoine.

Exercice 6.4

Lancez *Vim* et collez-y le texte ci-dessous, puis complétez les lignes du dessus en les faisant correspondre avec celles du dessous (commande d'ajout a). Supprimez ensuite les lignes du dessous (commande de suppression de ligne dd).

Lagardère la tête.

Lagardère secoua la tête.

- J'aimerais mieux Carrigue et mes gens avec leurs , répliqua-t-il.
- J'aimerais mieux Carrigue et mes gens avec leurs carabines, répliqua-t-il.

Il s'interrompit tout à pour demander :

Il s'interrompit tout à coup pour demander :

- Êtes-vous venu ?
- Êtes-vous venu seul ?

- Avec un enfant : , mon page.
- Avec un enfant : Berrichon, mon page.

- Je le connais ; il est leste et . S'il était possible de le faire venir...
- Je le connais ; il est leste et adroit. S'il était possible de le faire venir...

Nevers mit ses doigts entre ses lèvres, et donna un coup de retentissant.

Nevers mit ses doigts entre ses lèvres, et donna un coup de sifflet retentissant.

Un coup de sifflet pareil lui répondit derrière le de la Pomme-d'Adam.

Un coup de sifflet pareil lui répondit derrière le cabaret de la Pomme-d'Adam.

- La question est de savoir, murmura , s'il pourra parvenir jusqu'à nous.
- La question est de savoir, murmura Lagardère, s'il pourra parvenir jusqu'à nous.
- Il passerait par un trou d' ! dit Nevers.
- Il passerait par un trou d'aiguille ! dit Nevers.

Exercice 6.5

1. Lancez *Vim* puis créez le fichier « fic1.txt » qui contiendra uniquement un petit texte de quelques lignes. Enregistrez-le.

2. Enregistrez le fichier sous le nom « `fic1_2.txt` » et remplacez quelques caractères minuscules en majuscules (cette opération peut être réalisée à l'aide de commandes *Vim*). Enregistrez le fichier.
3. Ouvrez à nouveau « `fic1.txt` » et scindez la fenêtre de *Vim* verticalement de manière à visualiser les deux fichiers côte à côte.
4. Visualisez les différences entre les fichiers.
5. Transformez l'ensemble du texte de « `fic1.txt` » en majuscules. Revenez à l'état initial.

Exercice 6.7

Depuis l'invite de commande du terminal, lancez le programme `vimtutor` qui est un tutoriel interactif.

7 Interpréteur de commandes

« It is easier to port a shell than a shell script. » Larry Wall.

7.1 Introduction à Unix et aux interpréteurs de commandes

Unix est un système d'exploitation multitâches et multi-utilisateurs, conçu pour gérer les ressources d'un ordinateur de manière efficace. Comme l'illustre la figure 7.1, le système est structuré en couches successives autour d'un noyau logiciel (*kernel*) qui gère les interactions avec le matériel. Les utilisateurs interagissent avec le système à travers des applications et des commandes qui reposent sur des couches de plus haut niveau, offrant une abstraction croissante du matériel sous-jacent.

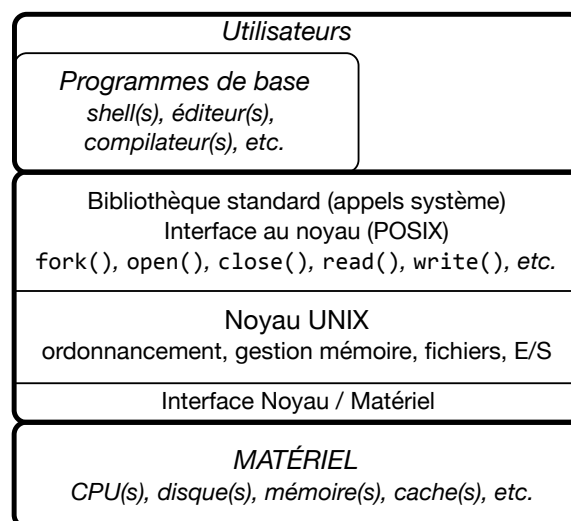


Fig. 7.1 : Architecture en couches d'Unix.

L'interpréteur de commandes, communément appelé *shell*, est un programme qui lit les commandes entrées par l'utilisateur, les interprète, puis les exécute. Il offre également un langage de script qui permet d'automatiser des tâches répétitives. Le *shell* est donc à la fois une interface utilisateur et un outil de programmation.

7.2 Différentes familles d'interpréteurs de commande

Les interpréteurs de commande se déclinent en plusieurs familles, dont les plus courantes sont la famille Bourne et la famille C.

La famille Bourne, qui tire son nom du Bourne Shell (*sh*), inclut des variantes comme *ksh* (*Korn shell*), *bash* (*Bourne again shell*), et *zsh*. Ces interpréteurs sont largement utilisés en raison de leur compatibilité avec les scripts Unix traditionnels et leur flexibilité.

La famille C, quant à elle, comprend des interpréteurs comme `csh` (*C shell*) et `tcsh` (*Tenex C shell*). Ces interpréteurs se distinguent par une syntaxe inspirée du langage C, ce qui les rend plus familiers aux programmeurs C, mais ils sont moins utilisés dans les scripts systèmes.

Pour déterminer quel interpréteur est actuellement utilisé, vous pouvez interroger la variable d'environnement « `SHELL` », qui stocke le chemin d'accès à l'interpréteur en cours d'utilisation. Par exemple, en exécutant la commande suivante :

```
$ echo $SHELL
/bin/bash
```

Ceci affichera le chemin d'accès à l'interpréteur, tel que `/bin/bash` si vous utilisez l'interpréteur `bash`.



Un nouvel interpréteur peut être lancé depuis le terminal en indiquant son nom. Par exemple `sh` lancera un interpréteur *Bourne*, `bash` un interpréteur *Bourne again* et `$SHELL` l'interpréteur par défaut. Pour l'arrêter, on utilise la commande `exit` ou la séquence de touches `Ctrl + d`.

7.3 Fichiers de configuration

7.3.1 Fichiers de connexion

Lorsqu'un utilisateur ouvre une session sur un système Unix, une série de fichiers de configuration sont exécutés afin d'initialiser l'environnement. Le fichier global `/etc/profile` est le premier à être lu, appliquant des configurations générales à tous les utilisateurs. Ensuite, ce sont les fichiers de configuration spécifiques à l'utilisateur qui sont lus dans cet ordre : `$HOME/.bash_profile`, `$HOME/.bash_login`, ou `$HOME/.profile`¹. Lorsque l'utilisateur se déconnecte, le fichier de déconnexion `$HOME/.bash_logout` est exécuté pour permettre des actions de nettoyage ou de sauvegarde.

Le fichier `.profile` est à utiliser de préférence parce qu'il est compatible avec tous les interpréteurs de la famille Bourne, tandis que `.bash_profile` est spécifique à `bash`.

7.3.2 Contenu typique d'un fichier `.profile`

Le fichier `.profile`, ou ses équivalents comme `.bash_profile` ou `.bash_login`, définit des variables d'environnement qui configurent le comportement de l'interpréteur pour la session de l'utilisateur. Par exemple, la variable `PATH` spécifie les dossiers dans lesquels l'interpréteur doit rechercher les exécutables. `LD_LIBRARY_PATH` indique les dossiers contenant les bibliothèques dynamiques nécessaires aux programmes, tandis que `MANPATH` spécifie où trouver les pages de manuel du système.

L'invite de commande est configurée par les variables `PS1` (invite primaire) et `PS2` (invite secondaire), qui déterminent respectivement l'apparence de l'invite lorsqu'une nouvelle commande est attendue et lorsqu'une commande multi-lignes est en cours de saisie.

Pour appliquer ces configurations à l'ensemble des sous-processus de l'interpréteur, il est nécessaire d'utiliser la commande `export` pour chaque variable, comme dans l'exemple suivant :

```
export PATH LD_LIBRARY_PATH MANPATH PS1 PS2
```

¹Si l'un des fichiers de configuration, tels que `.bash_profile` ou `.profile`, est absent, le *shell* passe simplement au fichier suivant dans l'ordre de préférence. Cela garantit que l'environnement de l'utilisateur est configuré même en l'absence de certains fichiers spécifiques.

Exemple de modification de l'invite de commandes

```
$ PS1=">> "
$ export PS1
>>
>> export PS1=`date` " > "
Mer 27 mai 2020 14:51:12 CEST >
Mer 27 mai 2020 14:51:12 CEST > export PS1="\w > "
/home/jsaigne/cours/odl >
/home/jsaigne/cours/odl > export PS1="\W > "
odl >
odl > export PS1="$ "
$
```

7.3.3 Fichier `.bashrc` (bash)

Le fichier `.bashrc` est exécuté à chaque fois qu'un nouvel interpréteur est lancé, comme lorsqu'une nouvelle fenêtre de terminal est ouverte. Ce fichier est souvent utilisé pour personnaliser l'environnement de travail en définissant des *alias* (raccourcis pour des commandes plus longues), en configurant des options spécifiques à l'interpréteur, et en surchargeant les paramètres définis dans le fichier `.profile`.

Par exemple, vous pouvez définir un *alias* pour afficher les fichiers d'un dossier avec des détails supplémentaires en ajoutant la ligne suivante dans `.bashrc` :

```
alias ll='ls -la'
```

La suppression d'un *alias* nécessite l'utilisation de la commande interne `unalias`.

Si vous souhaitez démarrer un interpréteur bash sans charger ce fichier, vous pouvez utiliser l'option `--norc` lors du lancement de bash.

7.4 Gestion des commandes et historique

L'interpréteur utilise un mécanisme appelé `readline`² (éditeur de commande) afin de gérer les commandes saisies par l'utilisateur. Ces commandes sont stockées dans un fichier d'historique, ce qui permet de rappeler facilement les commandes précédentes à l'aide de raccourcis clavier. Par exemple, `Ctrl` + `p` permet de rappeler la commande précédente, tandis que `Ctrl` + `n` affiche la commande suivante dans l'historique.

La taille de cet historique est contrôlée par la variable `HISTSIZE`, que vous pouvez configurer pour spécifier le nombre de commandes à retenir. Par exemple, pour définir un historique de 100 commandes, vous pouvez ajouter la ligne suivante dans votre fichier de configuration :

```
HISTSIZE=100
```

La taille de l'historique est indiquée dans la variable `HISTSIZE`. Par exemple `HISTSIZE=100`.



- `Ctrl` + `c` : annule la commande en cours;
- `Ctrl` + `n` sous bash : réalise la complétion des noms de fichiers ou des commandes.

²L'éditeur de commande `readline` est commun à la plupart des interpréteurs de commandes, ainsi qu'à l'éditeur *Emacs*.

La commande *shell* `history` permet d'afficher la totalité des commandes entrées.

7.5 Exécution des commandes

7.5.1 Structure d'une commande

Une commande dans le *shell* est composée du nom de la commande suivi d'éventuelles options, d'arguments, et de redirections. Une commande typique pourrait ressembler à ceci :

```
$ commande [-options] [arguments] [< inputfile] [> outputfile] [&]
```

Ici, les options modifient le comportement de la commande, les arguments fournissent des données à traiter, et les redirections permettent de spécifier des fichiers pour l'entrée et la sortie. Le caractère « & » à la fin d'une commande la fait s'exécuter en arrière-plan, ce qui permet de continuer à utiliser le terminal pendant que la commande s'exécute.

7.5.2 Exécution séquentielle

Les commandes peuvent être exécutées les unes après les autres, en attendant que chaque commande soit terminée avant de passer à la suivante :

```
$ commande1  
$ commande2
```

Il est également possible de combiner plusieurs commandes sur une seule ligne en les séparant par des points-virgules :

```
$ commande1 ; commande2
```

Auquel cas, la commande `commande1` doit être terminée avant d'exécuter `commande2`.

7.5.3 Exécution parallèle

Pour exécuter des commandes en parallèle, vous pouvez utiliser le caractère « & » à la fin de la commande. Cela signifie que la commande s'exécute en arrière-plan et que le *shell* est immédiatement prêt à en recevoir de nouvelles :

```
$ commande1 & commande2
```

Dans l'exemple suivant, les commandes `commande1` et `commande2` sont aussi exécutées en parallèle :

```
$ commande1 &  
$ commande2
```

`commande1` est exécutée en parallèle avec le `bash` : l'invite de commande revient immédiatement et l'on peut à nouveau entrer des commandes, puis `commande2` est exécutée en même temps que `commande1`.

Par exemple, vous pouvez lancer un téléchargement long en arrière-plan tout en continuant à utiliser le terminal pour d'autres tâches :

```
$ wget http://www.exemple.fr/fichier.zip &  
$ echo "Le téléchargement se poursuit en arrière-plan."
```

Ici, la commande `wget` télécharge le fichier en arrière-plan, tandis que vous pouvez continuer à utiliser le terminal pour d'autres commandes.

7.5.4 Exécution conditionnelle

Le *shell* permet d'exécuter des commandes conditionnellement. Une exécution conditionnelle peut être une conjonction (« et » / « && ») ou une disjonction (« ou » / « || »). Par exemple, la commande suivante n'exécute `commande2` que si `commande1` réussit (c'est-à-dire qu'elle retourne un code de sortie égal à 0) :

```
$ commande1 && commande2
```

À l'inverse, la commande suivante exécute `commande2` seulement si `commande1` échoue (retourne un code de sortie non nul) :

```
$ commande1 || commande2
```



Pour rappel, la valeur de retour d'une commande est la valeur retournée par la fonction `main()` en langage C (les commandes Unix sont écrites en C).

7.6 Gestion des entrées-sorties et redirections

Sous Unix, toutes les entrées et sorties sont considérées comme des **fichiers**. Par défaut, trois fichiers sont ouverts : `stdin` (entrée standard, habituellement associée au clavier), `stdout` (sortie standard, habituellement associée à l'écran), et `stderr` (sortie d'erreur, également dirigée vers l'écran).

7.6.1 Redirections

Il est possible de rediriger la sortie d'une commande vers un fichier avec le caractère `>` ou d'utiliser un autre fichier comme source d'entrée avec `<` :

```
$ commande > fichier_sortie # Redirige stdout vers un fichier
$ commande < fichier_entrée  # Redirige stdin depuis un fichier
$ commande 2> fichier_erreur # Redirige stderr vers un fichier
```

Pour rediriger simultanément `stdout` et `stderr` vers un même fichier³, vous pouvez utiliser la syntaxe suivante :

```
$ commande > sortie_et_erreur.txt 2>&1
```

³Lorsque l'on redirige la sortie d'une commande vers un fichier, il est nécessaire de s'assurer que l'on possède les permissions d'écriture sur le fichier de sortie. Si les permissions sont insuffisantes, la redirection échouera et le fichier ne sera pas modifié.



La commande `exec` est utile lorsque l'on souhaite rediriger les entrées ou sorties de façon permanente au sein d'un script ou d'une session de *shell*. Par exemple, vous pouvez rediriger toutes les sorties d'une session vers un fichier journal afin de capturer les événements d'une opération longue :

```
$ exec >logfile.txt 2>&1
```

Il est possible de réaliser l'affectation constante d'une entrée ou d'une sortie à un numéro qui correspond au *descripteur* d'un fichier :

- **exec n<fichier**. Par exemple : « `exec 5<&0` » lie le descripteur 5 avec l'entrée standard ;
- **exec n>fichier**. Par exemple : « `exec 6>toto.txt` » lie le descripteur 6 avec le fichier « `toto.txt` ».

7.6.2 Pipelines

Un **tube** permet d'utiliser la sortie d'une commande comme entrée d'une autre commande, le tout sans passer par un fichier. Un tube est mis en oeuvre à l'aide du caractère « `|` » :

```
$ commande1 | commande2
```

Plusieurs commandes peuvent être mises bout à bout à l'aide de tubes de manière à former des **pipelines** de commandes.

Pour enregistrer la sortie intermédiaire dans un fichier tout en continuant le pipeline, vous pouvez intercaler la commande `tee` entre les commandes :

```
$ commande1 | tee fichier_intermediaire | commande2
```

7.7 Variables du *shell*

Les variables peuvent être utilisées pour stocker et réutiliser des valeurs tout au long d'un script.

7.7.1 Affectation

```
$ variable=valeur
```

Exemple : `valeur=12`



Le *shell* traite toutes les valeurs comme des chaînes de caractères, même les nombres.

Exemples :

- `dossier_de_travail="/home/jsaigne/projet"`
- `valeur=12+3`
- `echo $valeur` : retourne 12+3 (et non pas 15)
- `nom=fichier`
- `echo ${nom}.txt` : retourne `fichier.txt`

7.7.2 Désaffectation

```
$ unset variable
```

7.7.3 Récupérer la valeur de la variable

On récupère le contenu d'une variable avec `$variable` ou `${variable}`, mais pas `$(variable)`.

Préférez `${variable}` quand il y a plus d'une lettre ou lorsque vous souhaitez concaténer le contenu avec autre chose : `${variable}xxx`.



Ne pas confondre avec `{commande}` !

7.7.4 Variables système

Le *shell* définit plusieurs variables système importantes, telles que `$?`, qui contient le code de retour de la dernière commande exécutée, et `$$`, qui contient le numéro de la dernière commande exécutée. Ces variables sont particulièrement utiles dans les scripts pour contrôler le flux d'exécution et éviter les conflits de noms dans les fichiers temporaires.

7.7.5 Portée des variables : variables d'environnement

Les variables sont locales au *shell* sauf si elles sont explicitement exportées (commande « `export` »).

```
$ export var
```

Dans ce cas-là, on exporte `var` et `$var` qui est son contenu.

Les variables d'environnement sont des variables spéciales qui peuvent être héritées par les processus enfants du *shell*. Elles sont généralement définies en majuscules et sont exportées avec la commande `export` :

Des variables comme `PATH`, `LD_LIBRARY_PATH`, et `MANPATH` sont essentielles pour le fonctionnement du système, car elles définissent les dossiers où le *shell* doit rechercher les exécutables, les bibliothèques, et les pages de manuel.

La commande *shell* `export` liste les variables d'environnement lorsqu'elle n'a aucun argument.

7.7.6 Substitutions

Le *shell* offre plusieurs mécanismes de substitution pour manipuler les variables de manière dynamique :

- `${var}` : identique à `$var` (sert à la concaténation).
- `${var-valeur}` : si `$var` est définie rend `$var` sinon rend `valeur`.
- `${var=valeur}` : si `$var` n'est pas définie affecte `$var` à `valeur` -- dans tous les cas rend `$var`.
- `${var?message}` : si `$var` est définie rend `$var` sinon affiche le message.
- ``commande`` : exécution de la commande.

Cas particulier de `bash` : « `$((1+2))` » retourne 3 (gère uniquement des entiers).

7.8 Fichiers de scripts *shell*

Un **script** *shell* est un fichier texte contenant une série de commandes *shell* à exécuter. Pour rendre un script exécutable, il est nécessaire d'ajouter l'autorisation d'exécution avec la commande « `chmod +x` » suivie du nom du script.

7.8.1 Entête

Les commentaires monolignes commencent en *shell* par le caractère « `#` ».

Sur la première ligne du script, il est possible d'indiquer le choix de l'interpréteur pour l'exécuter :

```
#!/bin/sh
#!/bin/bash
#!/usr/bin/perl
#!/usr/bin/python
```

Dans le cas contraire, il faut exécuter le script avec l'interpréteur :

```
$ sh ./essai.sh
```

Si un script *shell* est exécuté sans entête (la ligne « `#!/bin/bash` par exemple), le script sera interprété par le *shell* courant. Cela peut entraîner des comportements inattendus si le script est conçu pour un autre *shell*.

Afin de lister les commandes au fur et la mesure de leur exécution, il est possible d'ajouter l'option `-x`.

7.8.2 Arguments (de script ou de fonction)

Les paramètres de position spécifiques des scripts *shell* sont :

- `##` : nombre d'arguments (voir `argc`).
- `*` : liste de toutes les valeurs de paramètres dans une seule chaîne.
- `@` : liste de toutes les valeurs de paramètres dans une seule chaîne. Attention, le retour est différent de `*` :
 - « `$@` » : « `$1` », « `$2` », etc.;
 - « `$*` » : « `$1 $2 ...` ».
- `$0` : nom du programme ou de la fonction (voir `argv[0]`).
- `$1`, `$2`, ..., `$9` : valeur de l'argument *i* (voir `argv[1]`, `argv[2]`, etc.).



La commande `shift` décale la numérotation des arguments : `$2`, devient `$1`, etc.

- Rendre exécutable un script : `chmod 700 script`.
- Exécuter un script avec arguments : `./script a1 a2 a3`.

7.9 Commande `test`

Les tests peuvent être réalisés à l'aide de la commande « `test` » du *shell*, ou encore en utilisant les métacaractères « `[` » et « `]` » (il faudra alors faire attention aux espaces après `[` et avant `]`).

Cette commande retourne 0 en cas de succès ou une valeur supérieure à 0 dans le cas d'une erreur.

7.9.1 Tests numériques

Syntaxe

```
$ test valeur1 option valeur2
```

où option peut prendre les valeurs : `-eq` (égal), `-ne` (différent), `-gt` (strictement supérieur), `-ge` (supérieur ou égal), `-lt` (strictement inférieur) ou `-le` (inférieur ou égal).

Exemple

```
test 5 -lt 6 (équivalent à "[ 5 -lt 6 ]") retourne 0.
```

7.9.2 Tests sur les fichiers

Syntaxe

```
$ test option fichier
```

où option peut prendre les valeurs :

- `-s` : le fichier n'est pas vide (ex. : `test -s script`);
- `-d` : le fichier est un dossier (différent d'un fichier);
- `-f` : le fichier est un fichier (différent d'un dossier);
- `-w` : le fichier a les droits d'écriture;
- `-r` : le fichier a les droits de lecture;
- `-x` : le fichier a les droits d'exécution.

Exemples

```
$ test -d .
$ echo $?
0                                # -> vrai
$ test -f .
$ echo $?
1                                # -> faux
$ test -f scriptshell.sh
$ echo $?
0                                # -> vrai
$ test -x scriptshell.sh
$ echo $?
1                                # -> faux
$ chmod u+x scriptshell.sh
$ test -x scriptshell.sh
$ echo $?
0                                # -> vrai
```

7.9.3 Tests sur les chaînes de caractères

Syntaxe n°1

```
$ test chaîne option chaîne*
```

où option peut prendre les valeurs : « = » ou « != ».

Syntaxe n°2

```
$ test option chaîne
```

où option peut prendre les valeurs : -z (longueur nulle), -n (longueur non nulle).

Exemples

```
$ test -z " "
$ echo $?
1                               # -> faux
$ a="ensicaen" ; test -z $a
$ echo $?
1                               # -> faux
$ test -z ""
0                               # -> vrai
$ test -n "essai"
0                               # -> vrai
```

7.9.4 Opérateurs logiques

- **! test** : vrai si le test est faux.
- **test -a test** : ET logique entre test
- **test -o test** : OU logique entre test

Exemples

```
$ test -z "" -o -n ""
$ test ! -z ""
```

7.10 Structures de contrôle

7.10.1 Structure if

La structure if permet d'exécuter des commandes conditionnellement.

Syntaxe

```
if [ condition ]; then
    # Instructions si la condition est vraie
else
    # Instructions si la condition est fausse
fi
```

Attention : chaque partie de la structure est séparée par une fin de ligne ou par un point-virgule « ; ».

Exemple 1

```
if commande; then commande; else commande; fi
```

Le test du if porte sur le code de retour de la commande (\$?).

Variante

```
if ... then ... elif then ... fi
```

Exemple 2

```
a=1
if [ $a -eq 1 ]; then echo "a vaut 1"; else echo "a ne vaut pas 1"; fi
```

7.10.2 Structure case

La structure case permet de comparer une variable à plusieurs valeurs possibles et d'exécuter des commandes en fonction de la correspondance.

Syntaxe

```
case $variable in
    valeur1) commande ;;
    valeur1) commande ;;
    valeur1) commande ;;
    *) commande ;;
esac
```

```
case $variable in
    cas1) commande* ;;
    casn) commande* ;;
    *) commande ;;
esac
```

où *valeur1* est un motif qui correspond à une valeur pour l'entrée, en principe une constante ou une expression construite avec les métacaractères.

Exemple

```
#!/bin/bash
echo -n "Entrer un nombre 1 < x < 10 : "
read x
case $x in
    2|3|4|5|6|7|8|9) echo "x = $x";;
    *) echo "Mauvais choix !";;
esac
```


7.10.3 Boucle while

La boucle while exécute des commandes tant qu'une condition reste vraie.

Syntaxe

```
while [ condition ]; do
    # Instructions
done
```

La boucle est exécutée tant que le code de retour de la dernière commande est 0. Il y a possibilité d'utiliser les instructions break et continue.

Exemples

Menu

```
#!/bin/bash
clear ; boucle=y
while [ $boucle = y ]
do
    echo "Menu"; echo "===="
    echo "D: Affiche la date"
    echo "U: Affiche les utilisateurs connectés"
    echo "T: Affiche le répertoire de travail"
    echo "Q: quitter"
    echo
    read -s choix # mode silencieux : aucun écho sur le terminal
    case $choix in
        D|d) date ;;
        U|u) who ;;
        T|t) pwd ;;
        Q|q) boucle=n ;;
        *) echo "Choix impossible" ;;
    esac
    echo
done
```

Boucle infinie

```
while [ 1 ]; do echo "Bonjour"; done
```

7.10.4 Boucle for

La boucle for itère sur une liste de valeurs et exécute des commandes pour chaque valeur.

Syntaxe

```
for var in liste; do
    # Instructions
done
```

liste est une liste de valeurs que doit prendre la variable var.

Exemples

```
#!/bin/bash
for fruits in oranges pommes poires
do
    echo "Je mange des $fruits"
    sleep 1
done

$ for i in 1 2 3 4 5; do; echo $i; done!
```



Il est possible d'utiliser les instructions `break` et `continue`.

7.11 Fonctions

Les fonctions permettent de rassembler un ensemble de commandes sous un même nom, ce qui facilite la réutilisation du code.

Syntaxe de la déclaration

```
[function] nom_fonction(){ }
```

7.11.1 Arguments

Les arguments passés à une fonction sont accessibles via `$1`, `$2`, etc., de la même manière que pour les scripts. La fonction retourne un code de sortie à l'aide de l'instruction `return`, sinon elle retourne le code de sortie de la dernière commande exécutée.

Les variables ne sont que globales et leur portée est limitée au *shell* qui exécute le script.

Exemple 1

```
test1() {
    echo "Nombre d'arguments $#"
```

```
    echo "Les arguments sont $*"
}
```

```
test1 a1 a2 a3
```

Exemple 2

```
if [ -f $x ]
then echo "Fichier"
else echo "Pas de fichier"
fi
```

7.11.2 Valeur de retour

L'instruction « `return n` » permet à une fonction de retourner un résultat numérique donné, sinon c'est le résultat de la dernière commande qui est retourné.

Le résultat est accessible par la macro `$?` (par ex. : « `echo $?` »).

7.12 Métacaractères du *shell*

7.12.1 Spécifier des ensembles de fichiers

- `*` : n'importe quelle suite de caractères même vide.
- `?` : un caractère exactement.
- `[abcde]` : n'importe quel caractère à l'intérieur de l'ensemble.



Le métacaractère est interprété **avant** l'appel de la commande.

Par exemple, la commande « `ls po*.c` » affiche tous les fichiers du dossier courant commençant par « `po` » et finissant par « `.c` ».

Au niveau des chaînes de caractères :

- `"xxx"` : la chaîne de caractères avec évaluation de son contenu (les variables sont remplacées et les `*` aussi).
- `'xxx'` : la chaîne de caractères sans évaluation du contenu :
 - « `echo "$HOME"` » donne « `/home/jsaigue` ».
 - « `echo '$HOME'` » donne « `$HOME` ».
- Le caractère « `\` » empêche l'interprétation particulière du caractère qui le suit (caractère de despéciation) :
 - « `echo "\$HOME"` » donne « `$HOME` ».
 - « `echo "\"un guillemet\""` » donne « `"un guillemet"` ».

7.12.2 Exécution de commandes

- ``` : retourne le résultat de l'exécution de la commande placée entre ``` et ```.
- `()` : exécution des commandes dans un sur-*shell*.
- `{ }` : exécution des commandes dans le même *shell*. Attention, ne pas confondre avec « `${xx}` ».

Exemple

```
$ PS1=`date`
```

7.13 Commandes spécifiques du *shell*

Nous rappelons certaines commandes que nous avons rencontrées et qui font partie intégrante de l'interpréteur de commandes :

- `cd`, `pwd`, `export`, `exit`, `ulimit`, `umask`, `set`, `unset`, `fg`, `bg`, `jobs`.
- `type commande` (ou `which <commande>`).
- `read variable` / `echo valeur` (ex. : `$ read a; echo $a`)
- `exec commande` : exécution de la commande à la place du *shell*.
- `eval commande` : évaluation de la commande (voir différence avec `{ }`)
- `trap`



La commande `trap` est essentielle pour gérer les interruptions ou les signaux envoyés à un script. Par exemple, si un script crée des fichiers temporaires, `trap` peut être utilisée pour s'assurer qu'ils sont supprimés même si le script est interrompu :

```
trap "rm -f /tmp/fichier_temporaire; exit" INT TERM EXIT
```

Cela garantit que les fichiers temporaires sont nettoyés lorsque le script se termine ou est interrompu par l'utilisateur (par exemple, avec `Ctrl + C`).

Exemple

```
if test $? -eq 0 && test -n "$oldtty"; then
    trap 'stty $oldtty 2>/dev/null; exit' 0 2 3 5 10 13 15
else
    trap 'stty $ncb echo 2>/dev/null; exit' 0 2 3 5 10 13 15
fi
```

7.14 Calculs en bash

Le *shell* ne connaît que les chaînes de caractères. Toutefois, il propose des extensions pour effectuer des calculs.

7.14.1 Opérations sur des entiers

```
a=$((1+2))
let b=4+3
somme=$((a+b))
echo $somme
```

7.14.2 Opérations sur des flottants

Pour effectuer des opérations sur des nombres flottants, vous pouvez utiliser le programme `bc` (*basic calculator*), qui agit comme une calculatrice en ligne. `bc` permet également de contrôler la précision des calculs, ce qui est particulièrement utile pour les calculs scientifiques. Par exemple, il est possible de configurer la précision à 10 décimales avec l'option `-l` :

```
a=$(echo "scale=10; 1.0/3.0" | bc -l);
```

La variable `a` contient un résultat avec 10.

7.15 Conclusion

Pour plus de détail sur l'usage de la ligne de commande, nous renvoyons le lecteur à l'ouvrage de W. E. Shotts [11].

Exercices

Exercice 7.1

En utilisant l'éditeur *Vim*, proposez un script qui récupère une phrase et, à l'aide de la commande `echo` et d'une redirection, stocke cette phrase dans un fichier texte.

Exercice 7.2

À l'aide de l'éditeur *Vim*, écrivez un script qui attend deux nombres, calcule leurs somme, différence, produit et rapport puis affiche les résultats.

Exercice 7.3

En utilisant l'éditeur *Vim*, proposez un script qui récupère le nom d'un dossier à archiver, puis qui initialise une variable avec le nom du fichier d'archive comprenant la date du jour, par exemple « `backup-25-12-2020.tar.gz` », et enfin crée l'archive.

Exercice 7.4

Sur Unix, « `/dev/random` » est un fichier spécial qui sert de générateur de nombres aléatoires.

Voici un exemple permettant de générer cinq propositions de mots de passe composés de dix caractères chacun :

```
$ cat /dev/urandom | tr -dc '!@#%&*()_A-Z-a-z-0-9' | fold -w10 | head -5
```

- `tr -dc '!@#%&*()_A-Z-a-z-0-9'` : indique les caractères à utiliser pour créer le mot de passe -- toutes les lettres de l'alphabet (majuscules/minuscules), tous les chiffres (de 0 à 9) et les caractères spéciaux « `!@#%&*()_` » sont utilisés;
- `fold -w10` : génère des mots de passes de dix caractères;
- `head -5` : extrait cinq mots de passe différents.

En utilisant l'éditeur *Vim*, écrivez le script « `pass_generator.sh` » qui prend comme arguments le nombre de mots de passe souhaités ainsi que leur longueur. Exemple d'appel pour générer 4 mots de passe de 8 caractères chacun :

```
$ crypt.sh 4 8
```

Exercice 7.5

Sachant que $\arctan(1) = \pi/4$, afficher la valeur de π en utilisant les commandes `echo`, `bc` ainsi qu'un tube.

8 Édition de programmes

« Good code is its own best documentation. » Steve McConnell.

Dans ce chapitre, nous supposons que le programme a été préalablement conçu par une analyse algorithmique. Nous allons nous intéresser au codage et plus particulièrement aux règles de mise en forme.

8.1 Introduction

8.1.1 Pourquoi respecter une mise en forme ?

Un programme est destiné à une double lecture : par une machine et par un humain; ces deux entités n'ont pas besoin de la même information. Pour ce qui est du décodage par une machine, la définition des différents langages précise les conditions dans lesquelles un programme peut être relu par celle-ci. Dans le cas de la lecture par un humain, ce sont les conventions de mise en forme qui améliorent la lisibilité du logiciel, permettant un gain de temps et de compréhension. Le but est aussi d'identifier au mieux les différentes parties du programme afin d'avoir une meilleure maîtrise de son contenu.

Il est en général difficile de comprendre ce que fait un programme à partir de son code, car il manque les objectifs et les motivations du code. C'est le rôle des commentaires d'**explicit**er la logique de conception du programme, le code n'étant que le résultat. Assez fréquemment, l'expression de la solution est éloignée de l'expression du problème, il est donc nécessaire d'essayer de garder le plus longtemps possible les termes du problème d'où l'insertion des commentaires associés au code. En effet, environ 80% du temps de mise au point va à la maintenance, car un programme est rarement maintenu par le même auteur au cours de sa vie.



Un programme sans commentaires ou avec des commentaires incomplets ou mal faits ne sert à rien, voire relève de la faute professionnelle pour un ingénieur en informatique. Les commentaires ne sont pas à faire une fois le code terminé, mais bien au cours de sa rédaction! En effet, la logique de conception étant connue avant l'écriture du code, il se peut qu'à tout moment, tout ou partie du code d'un programme puisse être partagé avec d'autres développeurs.

Il faut néanmoins éviter l'excès de commentaires qui irait à l'encontre de l'objectif!

Klaus Lambertz estime que la proportion de commentaires dans un fichier doit représenter **30 à 75 %** de la taille du fichier [6].

À présent, supposons qu'un concepteur facétieux mette en forme son programme tel que celui présenté ci-dessous. Bien que ce programme puisse présenter un certain esthétisme, certes subjectif, et que sa compilation s'effectue sans aucune erreur ni aucun avertissement, il n'en reste pas moins qu'il est difficile à déchiffrer. Outre le fait qu'il n'est pas documenté, sa réédition à l'aide d'un simple éditeur de texte pour y voir « plus clair » peut demander du temps. Il peut alors devenir judicieux d'utiliser un outil adapté pour le reformater.

```
/* eastman-2011.c */  
#include <stdio.h>
```

```
#include <math.h>
#include <unistd.h>
#include <sys/ioctl.h>

main() {
    short a[4];ioctl
    (0,TIOCGWINSZ,&a);int
    b,c,d=*a,e=a[1];float f,g,
    h,i=d/2+d%2+1,j=d/5-1,k=0,l=e/
    2,m=d/4,n=.01*e,o=0,p=.1;while (
    printf("\x1b[H\x1b[?251",!usleep(
    79383)){for (b=c=0;h=2*(m-c)/i,f=-
    .3*(g=(1-b)/i)+.954*h,c<d;c+=(b==
    b%e)==0)printf("\x1b[%dm ",g*g>1-h
    *h?c>d-j?b<d-c||d-c>e-b?40:100:b<j
    ||b>e-j?40:g*(g+.6)+.09+h*h<1?100:
    47:((int)(9-k+(.954*g+.3*h)/sqrt
    (1-f*f))+((int)(2+f*2))%2==0?10?
    :101);k+=p,m+=o,o=m>d-2*j?
    -.04*d:o+.002*d;n=(1+=
    n)<i||l>e-i?p=-p
    ,-n:n;}}
```



Le fichier « eastman-2011.c » est le code C d'un des lauréats du concours « The International Obfuscated C Code Contest » qui a lieu depuis 1984. Le site du concours est le suivant : <http://www.o.uconn.edu/ioccc.org/years.html>.

8.2 Écriture d'un programme

L'écriture d'un programme répond à un certain nombre de règles que nous décomposons en trois parties :

1. Règles de commentaire.
2. Règles de typographie et notamment d'indentation.
3. Règles de nommage des identificateurs.

8.2.1 Commentaires de programmes

Les conventions présentées ici sont consensuelles, car partagées par une grande part de la communauté des programmeurs. De plus, elles sont en cohérence avec le logiciel *Doxygen* présenté plus loin dans ce chapitre.

Introduction

Les commentaires utilisent `/**` ou `/*` pour distinguer deux types de commentaires : un commentaire de documentation ou un commentaire local au fichier.



Le commentaire monoligne `//`, bien que largement utilisé, ne respecte pas le standard ANSI et n'est donc pas portable.

Un bloc de commentaire s'écrit de la manière suivante :

```
/*          */
*          *
*          *
*/          */
```

8.2.1.1 Cartouches d'un programme {-}

Tout programme C est composé de quatre parties de commentaires de documentation appelées aussi « cartouches » :

1. Identification.
2. Présentation du contenu.
3. Gestion des versions.
4. Détail des fonctions.

Partie 1 : cartouche d'identification

Le but de cette partie est de situer le produit dans son environnement social : identification du programme pour son archivage, son identification et les restrictions liées à son utilisation (indépendante du programme).

Cet en-tête indique :

- l'entreprise où a été réalisé le programme et/ou les mécènes qui ont financé le travail (par exemple l'adresse de l'ENSICAEN);
- les *copyrights* ou droits d'utilisation, de modification et de diffusion (par exemple la licence : GPL, LGPL, Cecill, BSD). En Europe, un programme n'est pas soumis à la législation sur les brevets industriels, mais à celle du *copyright*. Aux États-Unis, il est possible de breveter un logiciel.

Exemple :

```
/*
* ENSICAEN
* 6 Boulevard Maréchal Juin
* F-14050 Caen Cedex
*
* This file is owned by ENSICAEN students. No portion of this
* document may be reproduced, copied or revised without written
* permission of the authors.
*/
```

Partie 2 : cartouche de présentation du contenu

Cette partie présente l'objectif du programme et de la solution utilisée.

Ce commentaire indique :

1. Nom du fichier : *@file*.
2. Résumé de la description du contenu : *@brief*.
3. Description détaillée de la solution préconisée.
4. Éventuellement : bibliographie, référence au cahier des charges ou les tests effectués.

Exemple :

```
/**
 * @file sort.c
 * @brief Sorts data in ascending order, using the insertion algorithm.
 *
 * Ref. : C. Froidevaux et al., "Algorithmes", Dunod, 1990.
 */
```

Partie 3 : cartouche de gestion des versions, des journaux et des bugs

Le but de cette partie est l'identification des auteurs et la description de l'état d'avancement du programme. Ces informations permettent de gérer les versions du programme (*versioning*) et d'aider à sa maintenance.

Ce commentaire indique :

1. Le nom des auteurs : `@author` (une étiquette par auteur).
2. Le numéro de version : `@version`.
3. Ce qui reste à faire : `@todo` (une étiquette par *item*).
4. Les *bugs* : `@bug` (une étiquette par *bug*).

Exemple :

```
/**
 * @author Auteur1 <mail@ecole.ensicaen.fr>
 * @author Auteur2 <mail@ecole.ensicaen.fr>
 * @version 0.0.1 - 2017-09-13
 *
 * @todo the list of improvements suggested for the file.
 * @bug the list of known bugs.
 */
```

Partie 4 : le cartouche des fonctions

Spécifier un sous-programme c'est préciser le contrat liant le concepteur et l'utilisateur. Dans ce contrat, le premier ignore ce qui sera fait du sous-programme et le second comment il est fait. Un sous-programme doit donc rester explicite même s'il est isolé de son environnement, cela afin de favoriser sa réutilisabilité.

Le cartouche indique :

1. La fonction réalisée par le sous-programme. Si celle-ci est trop longue, faire une référence.
2. Les arguments explicites sont ceux qui apparaissent dans la ligne de la fonction. Ils sont introduits par l'étiquette `@param`. On utilise une étiquette par paramètre.
3. La valeur de retour : `@return` que l'on ne spécifie pas si le type de retour est `void`.
4. Les arguments implicites sont les variables globales utilisées, les effets de bord, les valeurs en entrée ou en sortie. Il est nécessaire de les décrire dans le texte.
5. Le traitement des erreurs peut être intégré à l'aide des étiquettes `@bug`, `@todo`, `@version`, `@author`.

Exemple :

```
/**
 * Unmasks the data from the given mask. It means that the pixels are set to
 * the pixel value of the reference image when the related label in the mask
 * is 0.
```

```

* @param mask the region map that is used as a mask.
* @param reference the image that is used as a reference.
* @return 1 if the reference is successfully masked, 0 otherwise.
*/
int unmask(const int* mask, const int* reference) {
    /* ... */
    return 1;
}

```

Commentaires des types et des variables

Ces commentaires se placent avec les déclarations qu'ils précisent : soit regroupés dans un seul bloc, soit répartis sur chacune des variables. Le seul critère à prendre en compte est la lisibilité. Il s'agit ici uniquement de remplir les déclarations par d'autres caractéristiques que les constructions du langage.

```

int address; /* An address is a strictly positive even value. */
float price; /* Price is in euros. */

```

À exclure :

```

int address; /* address is an integer. */

```

Commentaires de code

Les commentaires de code sont à proscrire dans la mesure du possible. Il faut éviter au maximum de surcharger le code par des commentaires d'instruction. La majorité des commentaires doivent être mis dans le cartouche au-dessus de l'appel des fonctions.

Toutefois, les commentaires de codes sont nécessaires s'ils fournissent des précisions techniques comme dans les cas suivants :

- utilisation d'une astuce de programmation;
- détection d'une instruction qui provoque une erreur sans que l'on sache la corriger;
- indication d'une instruction à ne surtout pas modifier;
- toute remarque indispensable à la compréhension d'instructions complexes (mais pas de l'algorithme, ces commentaires étant mis dans le cartouche de la fonction).

Il peut être utile de mettre des commentaires de ponctuation en référence aux structures de contrôle, par exemple ouverture et fermeture de structures :

```

} /* end of while. */

```

Enfin, on évitera les commentaires de paraphrase :

```

i=0; /* reset i to 0 */

```

L'absurdité de ce pléonasme n'est malheureusement pas un frein à son emploi. Par contre, des commentaires comme : « la rame est remise à son terminus » ou « la baignoire est vidée » sont d'un apport certain.

Répartition entre fichiers « .h » et « .c »

Les entêtes contiennent tous les commentaires liés à la **communication entre développeurs** :

- le cartouche des fonctions permet de réutiliser la fonction;

- le cartouche des types permettant de construire des variables à partir de ces types.

Les fichiers « .c » contiennent tous les commentaires liés à la **maintenance du code** :

- le cartouche des fonctions permet d'indiquer les particularités techniques du code de la fonction ;
- les commentaires de code permettent de préciser les particularités des instructions.



En général, seuls les fichiers « .h » sont utilisés pour générer la documentation technique avec le logiciel *Doxygen*.

8.2.2 Mise en forme d'un programme

Longueur des lignes et césures

La longueur des lignes ne doit pas dépasser **80** caractères de manière à ne pas être trop dépendant de l'imprimante ou du terminal. Même si l'écran est large, il vaut mieux plusieurs fenêtres côte à côte que l'on pourra comparées, plutôt qu'une fenêtre très large.

Les règles de césure (*wrapping*) à appliquer sont les suivantes :

- après un point virgule ;
- après une virgule.

```
var = someMethod(longexpression1, longexpression2,
                  longexpression3);
```

- avant un opérateur. Par exemple :

```
value = x * (y + 3)
        + 4
```

Marquage des commentaires

Éviter les commentaires du style :

```
/*****
/*          Part A          */
*****/
```

Préférer :

```
/*
 * P A R T A
 */
```

En effet, du fait de leur longueur, ces lignes peuvent avoir des comportements différents selon les terminaux ou les imprimantes.

Conventions

a) Conventions de casse

- Variables : en minuscules.

- Constantes : en majuscules.
- Noms de fonction : en minuscules.

Puisque la plupart des langages n'autorisent pas l'espace dans les noms d'identificateurs, il est nécessaire de définir des conventions de séparation des mots. Dans notre cas, nous utilisons le tiret bas « _ » (notation *snake case*).

b) Convention de nommage des identificateurs

Le premier commentaire est le nom des identificateurs. Les déclarations du genre « `int sxfrt_tkj52;` » sont donc à proscrire. La plupart du temps, le nom des variables et des fonctions doit suffire à leur description. Par exemple, le commentaire ci-dessous devient inutile :

```
float capacitance; /* the capacitance */
```

En principe, chaque fonction effectue une action. Son nom doit indiquer clairement ce qu'elle fait (un *verbe*) et il peut être constitué de plusieurs mots. Par exemple :

```
check_for_errors();  
merge_data_to_file();  
get_max_retry();
```

Pour une constante on aura par exemple :

```
#define MAX_WATER_LEVEL 10
```

c) Instruction

Il faut se limiter à une instruction par ligne. C'est utile notamment pour la détection des erreurs puisque le numéro de ligne est affiché par le compilateur.

d) Déclaration

Il faut se limiter aussi à une déclaration par ligne afin de permettre les commentaires s'ils sont nécessaire.

```
int i; /* row index */  
int j; /* column index */
```

Mais il est possible de mettre plusieurs déclarations sur une même ligne, par exemple dans le cas d'indices de boucle :

```
int i, j;
```

Toutefois, il ne faut pas mélanger les types. Ce qui suit doit être évité et réparti sur trois lignes :

```
int i, *p, t[128];
```

e) Indentation

L'indentation permet de souligner la hiérarchie dans un code par ajout d'un décalage d'une tabulation à droite. Il existe deux standards internationaux qui permettent d'indenter les blocs (les autres pouvant ne pas être compris) :

<pre>/* à la Unix/C */ <u>void</u> f(type p) { instructions; }</pre>	<pre>/* à la Pascal */ <u>void</u> f(type p) { instructions; }</pre>
--	--

<pre>if (test) { instruction; }</pre> <pre>if (test) { instructions; } else { instructions; }</pre> <pre>while (test) { instructions; }</pre>	<pre>if (test) { instructions; }</pre> <pre>if (test) { instructions; } else { instructions; }</pre> <pre>while (test) { instructions; }</pre>
---	--



Il est préférable de toujours mettre des accolades pour les blocs, même s'il n'y a qu'une seule instruction.

Espaces

On insérera un espace après les mots clés, ainsi qu'avant l'accolade ouvrante d'un bloc ou d'une fonction :

```
for (i = 0; i < n; i++) {
    /* ... */
}

while (i == 0) {
    /* ... */
}
```

Par contre aucun espace ne sera inséré après le nom d'une fonction que ce soit lors de sa déclaration ou de son utilisation :

```
void f(int i) {
    /* ... */
}

f(10);
```



Attention : il n'y a en général jamais d'espace entre la fin d'une instruction et le « ; », sauf dans le cas d'une instruction *nulle* (ou *vide*).

Conclusion

Il ne faut en aucun cas rendre comme absolue et définitive la classification précédente, car elle ne prétend pas résoudre tous les cas. Elle n'a pour ambition que de permettre une meilleure compréhension de ce qui est utile à la transmission des programmes d'un individu à un autre.

8.2.3 Exemple de squelette de fichier C

```
/*
 * ENSICAEN
 * 6 Boulevard Maréchal Juin
 * F-14050 Caen Cedex
 *
 * This file is owned by ENSICAEN students.
 * No portion of this document may be reproduced, copied
 * or revised without written permission of the authors.
 */

/**
 * @file skeleton.c
 *
 * Description of the program objectives.
 * All necessary references.
 */

/**
 * @author Auteur1 <mail@ecole.ensicaen.fr>
 * @author Auteur2 <mail@ecole.ensicaen.fr>
 * @version 0.0.1 - 2017-09-13
 *
 * @todo the list of improvements suggested for the file.
 * @bug the list of known bugs.
 */

#include <stdio.h>
#include <strings.h>

/**
 * A complete description of the function.
 *
 * @param par1 description of the parameter par1.
 * @param par2 description of the parameter par2.
 * @return description of the result.
 */
int function1(int par1, char par2) {
    int var = 0;
    /* ... */
    return var;
}

/*
 *
 */
```

```

* Description of the command line if it uses arguments.
*/
int main(int argc, char* argv[]) {
    #define USAGE "usage : %s fichier_texte\n" /* message for French users */
    int i;

    if ((argc > 1) && (!strcasecmp("-h", argv[1]))) {
        fprintf(stderr, USAGE, argv[0]);
        return -1;
    }
    i = function1(3, 'c');

    return 0;
}

```



Nous vous conseillons de jeter un coup d'oeil au document « SEI CERT C Coding Standard » qui vous donnera un bon point de départ pour affiner les règles de présentation du code, mais aussi des bonnes pratiques de codage en C [1].

8.3 Formatage du code dans Vim

Artistic Style est une application multiplateforme qui permet de formater un code source écrit en langage C, C++, C# ou Java. L'outil est disponible à l'adresse <http://astyle.sourceforge.net>. Pour l'utiliser depuis *Vim*, il faut l'associer à l'option `formatprg`.

Par exemple, si dans le fichier « `.vimrc` », on ajoute la ligne suivante :

```
autocmd BufNewFile,BufRead *.c set formatprg=astyle\ -A14Hpj
```

alors on indiquera à *Vim* que l'outil de formatage est le programme `astyle` et qu'il sera utilisé sur tout nouveau tampon ou sur le tampon courant, ce sur les fichiers C, et en utilisant la règle de formatage « `A14Hpj` » :

- `A14` : style Google;
- `H` : espace après les mots-clés;
- `p` : espace autour des opérateurs;
- `j` : ajout d'accolades si nécessaire.

Soit le fichier suivant avant traitement par *astyle* :

```

int f(int x);

int main(){int a; if(f(3)==1){a=3;}return 0;}

int f(int x)
{
if(x<2)
return 1;
return 0; }

```

La commande « `gggqG` » permet d'appliquer le style au code source du tampon courant. On obtient alors :

```

int f(int x);

int main() {
    int a;
    if (f(3) == 1) {
        a = 3;
    }
    return 0;
}

int f(int x) {
    if (x < 2) {
        return 1;
    }
    return 0;
}

```

8.4 Génération de la documentation avec Doxygen

L'application *Doxygen* (<http://www.doxygen.org>) permet de générer automatiquement de la documentation sous forme HTML, RTF ou LaTeX¹.

La documentation est produite à partir des commentaires qui débutent par `/**` et utilisent les étiquettes `@param`, `@return`, etc.

Les étapes de production de la documentation sont les suivantes :

1. Générer un fichier de configuration par défaut : `doxygen -g [config_file]`. Sans précision du nom du fichier de configuration, la commande génère par défaut un fichier nommé « *Doxyfile* ».
2. Dans le fichier ainsi généré, il faudra configurer au minimum les lignes suivantes :

```

PROJECT_NAME      = "Mon projet qui changera le monde"
OUTPUT_DIRECTORY = dossier où créer les fichiers HTML
INPUT              = (liste des fichiers à inclure (nominatif))
FILE_PATTERNS     = *.h *.c (liste des fichiers à inclure par suffixe)
EXCLUDE            = liste des fichiers à exclure (nominatif)
GENERATE_HTML     = YES
GENERATE_LATEX    = NO
GENERATE_RTF      = NO

```

3. Produire la documentation à l'aide de la commande : « `doxygen Doxyfile` » ou « `doxygen config_file` ».

Ainsi, afin de produire puis de visualiser la documentation avec un butineur, par exemple *Firefox*, on réalisera les commandes suivantes :

```

#!/bin/bash
# doxygen -g
# vim Doxyfile
# mv Doxyfile maconfig
doxygen maconfig
firefox index.html &

```

¹LaTeX est le langage qui a en partie permis la mise en forme de ce cours. En fait, dans sa dernière version, le texte a été écrit en langage [Markdown](#) et nous utilisons l'outil [Pandoc](#) ainsi qu'un patron en LaTeX afin de produire le fichier PDF de sortie.

Exercices

Exercice 8.1

Indiquer au moins cinq infractions aux règles de codage dans l'exemple ci-dessous (numéro de ligne et problème constaté) :

```
/* malformed1.c */
int main(void){
    double stuv_wxyz[2][3]={0.23,0.34,0.45},{0.56,0.61,0.12}}, alpha = 0.23456;
    int i, j;

    for(i=0; i<2;i++)
    {
        for (j=0; j<3; j++) {
            stuv_wxyz[i][j] = (stuv_wxyz[i][j] < 0.5) ? stuv_wxyz[i][j] *
            alpha : stuv_wxyz[i][j];
        }
    }
    return 0;
}
```

Exercice 8.2

Indiquer au moins cinq infractions aux règles de codage dans l'exemple ci-dessous (numéro de ligne et problème constaté) :

```
/* malformed2.c */
int main (int argc, char **argv) {
    int xyzk[2][3]={1,2,3},{4,5,6}}, i, j=0;

    for(i=0; i< 2; ++i)
    {
        if (j==0){
            xyzk[i][j] = xyzk[i][j] -
            10;
            printf("%d \n", xyzk[i][j]);
        }
    }
    return 0;
}
```

Exercice 8.3

1. Lancez *Vim* et créez le fichier « header1.c » qui contiendra un cartouche d'identification. Enregistrez, puis fermez l'éditeur.
2. Lancez *Vim* et ouvrez le fichier « factorial.c » disponible sur l'exemplier. Insérez le fichier « header1.c » (cf. exercice précédent) au tout début du fichier, puis proposez les cartouches et commentaires manquants en anglais.
3. Dans un nouvel onglet, préparez un fichier « Doxyfile » et générez la documentation avec l'outil *Doxygen* directement depuis *Vim*.

Exercice 8.4

Formatez le fichier « `scattered_knight.c` » disponible dans l'exemplier, puis renommez sa version formatée en « `formatted_vim_knight.c` ».

9 Compilation

```
« #define QUESTION ((bb) || !(bb)) » Shakespeare.
```

Dans ce chapitre, nous passons en revue les différentes étapes de la compilation d'un programme écrit en langage C. Nous utilisons l'outil GCC (*GNU Compiler Collection*) qui intègre un certain nombre de compilateurs associés à des langages tels que le C, le C++, Objective-C, le Go, etc. Pour plus d'informations, le lecteur curieux pourra se reporter à la documentation officielle [12].

9.1 Étapes de compilation

La **compilation** regroupe plusieurs étapes, toutes réalisées à l'aide de la commande `gcc` :

- **précompilation**;
- **traduction**;
- **optimisation**;
- **assemblage**;
- **édition de liens**.

Ces différentes étapes nécessitent des fichiers en entrées et produisent des fichiers de sortie comme l'indique la figure 9.1 ainsi que le tableau suivant.

Étape	Fichier en entrée	Fichier en sortie
Précompilation	.c	.c
Traduction	.c	.s/.c
Optimisation	.c	.s
Assemblage	.s/.o	.o
Édition de liens	.o	exécutable

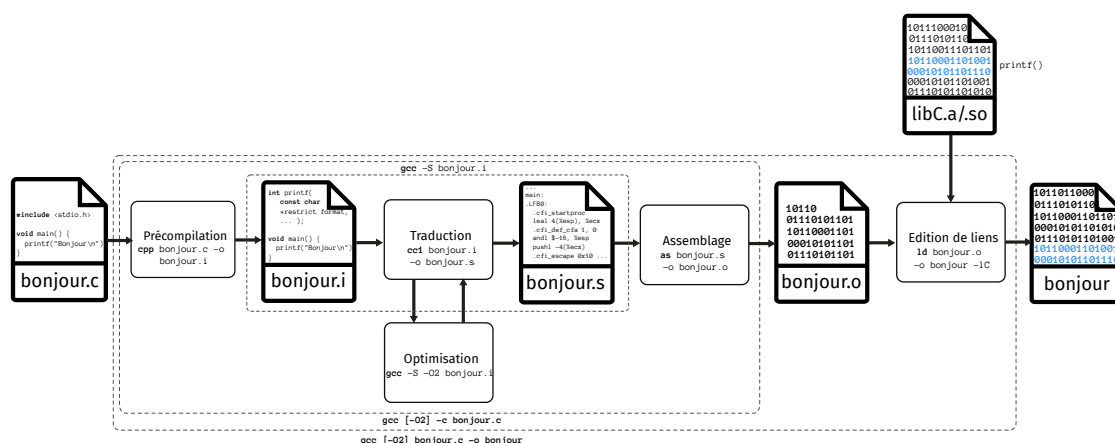


Fig. 9.1 : Chaîne de compilation permettant de produire l'exécutable « bonjour ».

GNU *gcc* est un « frontal » (*front-end* en anglais) qui réalise les différentes étapes de la compilation en appelant les outils adéquats en arrière-plan. La compilation avec GNU *gcc* est réalisée par l'appel « `gcc [-options] fichier.c` ».



Dans nos TP nous utilisons GNU *gcc*, mais il existe bien d'autres compilateurs C. Au nombre des plus connus, on trouve le compilateur *cc* (Unix), *clang* (Mac OS X), *CompCert* (INRIA), etc. En général le compilateur produit un code exécutable pour le microprocesseur de l'ordinateur sur lequel le compilateur est exécuté. Cependant il est possible d'utiliser un « cross-compilateur » afin de produire un code exécutable destiné à un autre type de microprocesseur.

9.2 Traduction

9.2.1 Rôle

L'étape de **traduction** consiste en la vérification de la syntaxe puis la génération du code intermédiaire, en général un code assembleur. Le code assembleur, quant à lui, est dépendant de la machine et optimisé pour elle.

9.2.2 Options de gcc

Dans le tableau ci-dessous, nous indiquons les quatre options utilisées lors de cette étape.

Option	Description
<code>-Wall, -Wextra</code>	Permettent l'affichage des avertissements (<i>warnings</i>). Des options plus spécifiques existent aussi par exemple <code>-Wunused-function</code> , etc.
<code>-ansi</code>	Compile du code ANSI ¹ (différent du code K&R d'origine). Ici les commentaires « <code>//</code> » sont détectés comme non ANSI. On se reportera à la référence sur le langage C de Kernighan et Ritchie [5].

¹American National Standards Institute.

Option	Description
-pedantic	Vérification de la norme ANSI. Pour la FSF (<i>Free Software Foundation</i>), l'utilisation de cette option est considérée comme un comportement pédant!

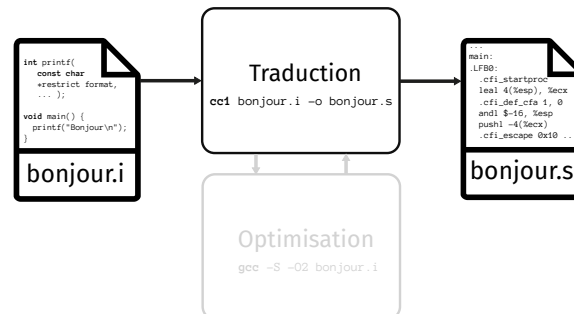


Fig. 9.2 : Chaîne de compilation : étape de traduction.



Dans ce cours nous produisons du code normalisé ANSI, c'est-à-dire qu'il respecte la norme du langage C dans sa version de 1989. Toutefois, le langage C propose des versions plus récentes apportant toutes leur lot d'améliorations. On trouvera par exemple les versions ISO/IEC C99 (1999) et même C11 ou C1X (2011). Afin de réaliser une compilation en prenant en compte ces versions, on remplacera l'option -ansi par « -std=c99 » ou « -std=c11 ».

9.3 Optimisation

9.3.1 Rôle

L'étape d'**optimisation** consiste à optimiser le code. Deux types d'optimisation coexistent : **en temps** et **en taille du code**. Par défaut, il n'y a aucune optimisation.

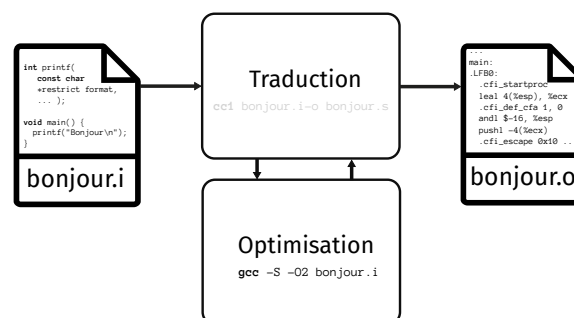


Fig. 9.3 : Chaîne de compilation : étape d'optimisation.

9.3.2 Options de gcc

GNU *gcc* propose les options suivantes correspondant à différents niveaux d'optimisation :

Option	Description
-O0	Aucune optimisation (utilisée lors de la phase de débogage).
-O ou -O1	Compromis temps / espace.
-O2	Optimisation en vitesse, en se permettant de prendre plus de place de mémoire. Par exemple, dupliquer le contenu des itérations en autant de lignes.
-O3	Encore plus de vitesse, en utilisant des astuces d'optimisation. Attention, cela peut changer le code (par exemple, changer l'ordre de lignes indépendantes pour rapprocher des lignes utilisant une même variable afin de la garder dans le registre).
-Os	Optimise la taille -- correspond à -O2 sans augmentation de taille.
-Og	Nouveau niveau d'optimisation général compatible avec le débogage.
-Ofast	Consiste en -O3 et -ffast-math.

À chaque niveau d'optimisation correspond un certain nombre d'opérations d'optimisation. Le tableau ci-dessous présente les différentes opérations d'optimisation proposées par GNU *gcc*, ainsi que les niveaux dans lesquels elles sont mises en oeuvre (le lecteur pourra se reporter à la documentation en ligne de GNU *gcc* pour une description plus complète des différentes optimisations : <http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html>).

Pour un niveau d'optimisation donné, le compilateur effectue un certain nombre de « passes » de manière à réaliser les différentes opérations d'optimisation. Ces passes résultent en plusieurs fichiers intermédiaires comme nous le verrons lors des séances de travaux pratiques.

9.3.3 Détail des optimisations

Il est possible de réaliser les opérations d'optimisation de manière individuelle en les appelant à l'aide du préfixe `-f`. Il est à noter que la réalisation ou non d'une opération d'optimisation dépendra de l'architecture utilisée.

Par exemple, l'optimisation `defer-pop` permet de ne jamais dépiler les arguments d'une fonction dès la fin de l'appel de la fonction (voir le principe d'empilement / dépilement des arguments de fonctions dans le cours de langage C). Elle est active par défaut pour tous les niveaux d'optimisation et aura pour effet de laisser les arguments s'accumuler sur la pile pour plusieurs appels de fonctions, puis de les dépiler tous en même temps. Pour l'activer de manière individuelle, il faudra entrer :

```
$ gcc -fdefer-pop -o prog prog.c
```

Le tableau ci-dessous regroupe les optimisations actives pour chacun des niveaux d'optimisation.

Optimisation	-O1	-O2	-Os	-O3
<code>defer-pop</code>	active	active	active	active
<code>merge-constants</code>	active	active	active	active
<code>thread-jump</code>	active	active	active	active
<code>cprop-registers</code>	active	active	active	active
<code>guess-branch-probability</code>	active	active	active	active
<code>omit-frame-pointer</code>	active	active	active	active

Optimisation	-01	-02	-0s	-03
loop-optimize	active	active	active	active
if-conversion	active	active	active	active
if-conversion2	active	active	active	active
delayed-branch	active	active	active	active
align-loops	inactive	active	inactive	active
align-jmps	inactive	active	inactive	active
align-labels	inactive	active	inactive	active
align-functions	inactive	active	inactive	active
optimize-sibling-calls	inactive	active	active	active
cse-follow-jumps	inactive	active	active	active
cse-skip-blocks	inactive	active	active	active
gcse	inactive	active	active	active
expensive-optimizations	inactive	active	active	active
strength-reduce	inactive	active	active	active
rerun-cse-after-loop	inactive	active	active	active
rerun-loop-opt	inactive	active	active	active
caller-saves	inactive	active	active	active
force-mem	inactive	active	active	active
peephole2	inactive	active	active	active
regmove	inactive	active	active	active
strict-aliasing	inactive	active	active	active
delete-null-pointer-checks	inactive	active	active	active
reorder-blocks	inactive	active	active	active
schedule-insns	inactive	active	active	active
schedule-insns2	inactive	active	active	active
inline-functions	inactive	inactive	inactive	active
rename-registers	inactive	inactive	inactive	active

9.4 Assemblage

9.4.1 Rôle

L'étape d'**assemblage** (*assembly*) consiste à transformer le code assembleur en un code exécutable (en réalité, *gcc* fait appel au programme *as* qui est un logiciel appelé « assembleur », comme le langage).



Fig. 9.4 : Chaîne de compilation : étape d'assemblage.

9.4.2 Options de gcc

Option	Description
-S	Retourne le programme en langage assembleur et crée un fichier de même nom suffixé « .s ».

9.5 Édition des liens

9.5.1 Rôle

L'étape d'**édition de liens** (*linking*) permet de construire le programme exécutable en réunissant tous les codes séparés en un seul module. Par exemple, il intègre dans le code du programme exécutable, le code objet des fonctions C, telles que celui de la fonction `printf()`. Ces dernières sont regroupées dans un fichier connu du compilateur, la bibliothèque dynamique « `libc.so` » dans le cas de Linux.

L'édition de liens fait appel au programme `ld`.



Dans le cas du système Ms-Windows et des DLL (*dynamic linked library*), c'est l'outil `dlltool.exe` qui devra être utilisé pour réaliser cette opération.

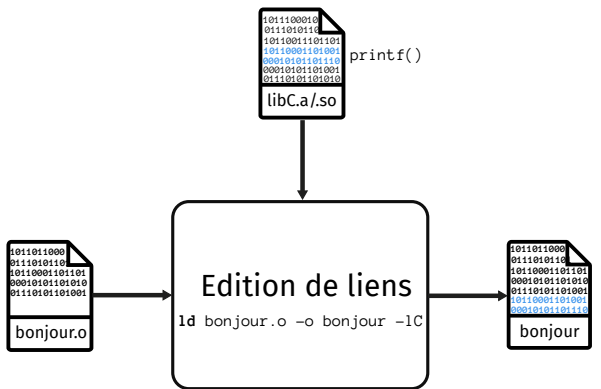


Fig. 9.5 : Chaîne de compilation : étape d'édition de liens.

9.5.2 Options de gcc

Option	Description
-o prog	Renomme le programme de sortie (par défaut le nom est « a.out »)
-c	Ne lance pas l'édition des liens et récupère un fichier « .o »
-llibrary	Lien avec la bibliothèque donnée (par exemple -lm)

En fait, la commande « gcc prog.c » est équivalente à la suite de commandes :

```
$ gcc prog.c -c
$ gcc prog.o -o a.out
```

Les commandes od et hexdump sous Unix permettent d'afficher le code d'un fichier binaire (fichier exécutable, fichier objet, etc.) sous sa forme hexadécimale.

```
$ od -x prog
$ od -t c -t x1 prog
$ hexdump -C prog
```

9.6 Précompilation

9.6.1 Rôle

L'étape de précompilation supprime les commentaires (`/* */`), puis interprète toutes les directives de compilation (introduites par le métacaractère « `#` »). Elle produit un fichier C sans commentaires et sans directives de compilation.

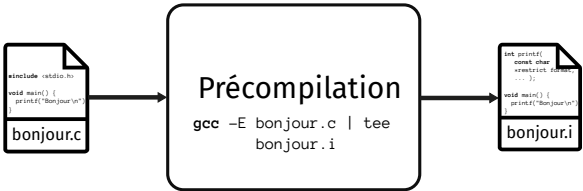


Fig. 9.6 : Chaîne de compilation : étape de précompilation.

L'étape de précompilation fait appel au programme `cpp` (*C pre-processor* ou précompilateur en français).

9.6.2 Option(s) de gcc

Option	Description
-E	Retourne le code précompilé seulement (sur la sortie standard).

9.6.3 Directives `#include`

```
#include <file_header.h>
```

Le fichier est recopié tel quel dans le fichier destination. Un fichier inclus peut contenir d'autres directives « `#include` », et à ce moment-là, il y a inclusion récursive.

Rôle d'un fichier d'entête

Le fichier d'entête contient toutes les définitions partageables entre fichiers C :

- toutes les définitions des types (redéfinition de type avec `typedef`);
- les constantes;
- les prototypes des fonctions utilisées dans le fichier qui inclut l'entête.

Le fichier d'entête est uniquement utilisé par le compilateur pour vérifier les appels des fonctions. Il n'inclut aucunement le code des fonctions qui y sont déclarées. Par exemple pour la fonction `printf()`, une compilation sans inclusion du fichier « `stdio.h` » considère que `printf()` prend comme paramètre un entier et retourne un entier. Il en est de même pour la fonction `pow()` déclarée dans « `math.h` ». Le compilateur va donc dans ces deux cas réserver de la place pour des entiers!



Il ne faut jamais mettre de chemin qui dépend de l'implantation particulière de chaque machine hôte, comme par exemple : « `#include "/usr/include/matrix.h"` ».

Écriture de code C

L'écriture de programmes C se compose de deux types de fichiers :

- fichiers « `.c` » qui contiennent le code des fonctions C;
- fichiers « `.h` » qui contiennent le prototype des fonctions définies dans les fichiers C et qui peuvent être utilisés par d'autres fichiers C (par exemple la fonction `printf()`).



Les fonctions qui n'ont pas à être partagées doivent être précédées du mot-clé `static` et ne pas être déclarées dans le fichier « `.h` ».

Répartition des commentaires

Les commentaires doivent être répartis ainsi :

- fichiers « `.h` » : contiennent les commentaires pour les échanges (réutilisation des fonctions);
- fichiers « `.c` » : contiennent les commentaires pour la maintenance (commentaires d'algorithmes et de code).

Appel

Concernant les inclusions de fichiers « `.h` », on peut noter la différence entre les appels :

- `#include <stdio.h>` où le fichier est recherché dans les dossiers du système (par ex. « `/usr/include` »);
- `#include "matrix.h"` où le fichier est d'abord recherché dans le dossier courant « `.` », puis dans les dossiers utilisateurs spécifiés par l'option `-I`.

Dans les deux cas, tous les dossiers sont essayés, mais c'est l'ordre de recherche qui est modifié. Si deux fichiers portent le même nom, c'est le premier trouvé dans l'ordre de recherche qui est pris en compte.

Options de gcc

Option	Description
-Ipathname	Ajoute un chemin vers des fichiers d'entête particuliers.

9.6.4 Directives #define

Ces directives permettent la définition de constantes et de macros.

- **Constantes** : remplacement littéral « #define NAME <expression> » :

```
#define NMAX 50
#define NMAX 10*5      mais      #define NMAX 50; -> erreur
```

- **Macro** : fonction sans type. Remplacement littéral suivi d'une capture des paramètres :

```
#define NAME(p) definition
#define CUB(x) x*x*x
#define MAX(x,y) (x>y)? x : y;
```

Options de gcc

Option	Description
-Dconstante=valeur	Remplace #define constant value, mais la directive #define reste prioritaire.
-Dconstante	Remplace #define constante, mais la directive #define reste prioritaire.

9.6.5 Directives conditionnelles #if

Elles permettent de définir différents codes à partir d'un même programme.

```
#if
/* ... */
#endif;

#if
/* ... */
#else
/* ... */
#endif;
#if
/* ... */
#elif
/* ... */
```

```
#endif;

#ifdef /* ... */
#ifndef /* ... */
#if (defined CONSTANT)
```

Exemple

Code spécifique pour Ms-Window :

```
#ifdef _WIN32
/* ... */
#endif
```

Afin d'éviter la multi-inclusion des fichiers entête, on placera **toujours** des directives d'exclusion dans ces fichiers entête. Par exemple pour le fichier « simulator.h » on écrira :

```
#ifndef SIMULATOR_H
#define SIMULATOR_H

/* déclarations du fichier simulator.h */

#endif
```

9.6.6 Directives #error et #warning

Les directives #error et #warning sont utilisées afin d'afficher des erreurs ou des avertissements au moment de la compilation. Par exemple :

```
#ifdef __WIN32
#error "Erreur système"
#endif
```

Elles servent aussi à rendre un fichier obsolète. Par exemple :

```
#warning "deprecated"
```

9.7 Compilation multifichiers

9.7.1 Notion de module

Il s'agit de décomposer le programme en plusieurs fichiers, chacun selon une préoccupation qui lui est propre. Cette décomposition permet entre autres de :

- éviter de tout recompiler (cf. programme de plusieurs dizaines de milliers de lignes);
- accroître la compréhension du programme (« diviser pour régner »);
- produire des fichiers réutilisables (« matrix.c », « fft.c », « image.c », etc.);
- faciliter le développement entre équipes différentes.

Il faut mettre le moins de code possible dans la fonction principale main().

9.7.2 Compilation modulaire

La compilation séparée consiste à s'arrêter avant l'édition des liens en générant les fichiers objets à partir des fichiers C. L'option utilisée est `-c`. Les codes objet forment le point de rencontre entre des modules qui peuvent provenir de langages différents.



Il est possible de renommer le code objet avec l'option `-o` (*output*).

Exemple

```
$ gcc file1.c -I../include -O2 -o ../objects/file1.o -c
```

9.7.3 Édition des liens

L'édition de liens des codes objet est obtenue par :

```
$ gcc *.o -o prog
```

On peut aussi utiliser la commande :

```
$ ld *.o -o prog -lc
```

En fait l'édition des liens avec `ld` n'est pas aussi simple et il vaut mieux laisser `gcc` s'occuper du passage des bonnes options. En lançant la commande « `ld *.o -o prog -lc` », il risque de se produire la sortie suivante :

```
ld : avertissement : le symbole d'entrée _start est introuvable ;  
utilise par défaut 00000000004002d0
```

et vous ne pourrez pas exécuter votre programme. Cet avertissement indique que l'éditeur de liens n'a pas pu trouver le point d'entrée `_start` du programme. Mais n'est-ce pas justement la fonction `main()` ce point d'entrée du programme? Oui, c'est bien le cas du point de vue du programmeur, par contre ça ne l'est pas du point de vue du système d'exploitation. Avant toute exécution d'un programme, il y a une « préparation du terrain » et du code binaire est ajouté au programme.

Alors, essayez ceci :

```
$ ld -static -o vector -L`gcc -print-file-name=`  
/usr/lib/x86_64-linux-gnu/crt1.o\  
/usr/lib/x86_64-linux-gnu/crti.o\  
vector.o /usr/lib/x86_64-linux-gnu/crtn.o\  
--start-group -lc -lgcc -lgcc_eh --end-group
```

Ou carrément : « `gcc *.c -o prog` » (`gcc -c` est alors implicite, puis est suivi de `gcc -o`).

Exemple

```
$ gcc ../objects/*.o -o ../bin/prog.exe
```

9.8 Bibliothèques

Une bibliothèque regroupe un ensemble de fichiers objets (« *.o ») sous un même nom. On distingue les bibliothèques statiques et les bibliothèques dynamiques.

Par exemple, l'utilisation de « `math.h` » implique l'option `-lm` lors de l'étape d'édition de liens, ce qui correspond aux fichiers « `libm.so` » (bibliothèque dynamique) ou « `libm.a` » (bibliothèque statique). L'appel de `gcc` se fera de la manière suivante :

```
$ gcc test_convolution.c -lm
```



La bibliothèque standard « `libc.so` » est incluse par défaut, donc il n'est pas nécessaire d'ajouter l'option `-lc`. Cette bibliothèque contient les fonctions courantes du langage C telles que `printf()`, `scanf()`, etc.

Il est important de bien faire la distinction entre le fichier d'en-tête, le(s) fichier(s) C associés et la bibliothèque proprement dite.

Par exemple, la fonction mathématique `pow()` a son prototype déclaré dans « `math.h` », son corps est défini dans le fichier « `w_pow.c` » fourni avec les codes sources de GNU `gcc` (non installés par défaut) et son code objet se trouve dans les bibliothèques « `libm.so` » et « `libm.a` ».

9.8.1 Compilation avec une bibliothèque

GNU `gcc` doit être appelé avec les options :

Option	Description
<code>-lname</code>	Inclut la bibliothèque « <code>libname.a</code> » ou « <code>libname.so</code> » (ou « <code>libname.dylib</code> » sur Mac OS X).
<code>-Lpathname</code>	Renseigne sur le dossier où se trouve la bibliothèque. Par défaut : « <code>-L/usr/lib</code> ».

9.8.2 Bibliothèque statique

Une bibliothèque **statique** regroupe des fichiers objets (« .o »). Elle est nommée « `lib<name>.a` » sous Linux (suffixe « `.lib` » sous Ms-Windows). L'inconvénient de ce type de bibliothèque est qu'il faut l'incorporer au code du programme lors de la phase d'édition de liens comme le montre la figure 9.1, ce qui entraîne une augmentation de la taille finale du code (programme + archive). Par contre, elle a pour avantage de fournir un programme « tout-en-un » comme l'illustre la figure 9.7.

9.8.3 Bibliothèque dynamique

Une bibliothèque **dynamique** regroupe des modules de codes exécutables. Elle ne sera liée qu'au moment de l'exécution. Nommée « `lib<name>.so` » sous Linux (« `lib<name>.dylib` » sous Mac OS X et « `.dll` » sous Ms-Windows). L'avantage de ce type de bibliothèque est que la taille finale du programme est moindre et qu'il peut

y avoir partage avec d'autres programmes (cf. figure 9.7) De plus, en cas de « correction » du code de la bibliothèque, tous les programmes qui l'utilisent en profitent. Par contre, elle nécessite de se trouver sur les machines hôtes --- combien de fois n'avez-vous pas pester sur l'absence de plusieurs fichiers « dll » lors de l'exécution d'un programme récupéré à droite à gauche ?

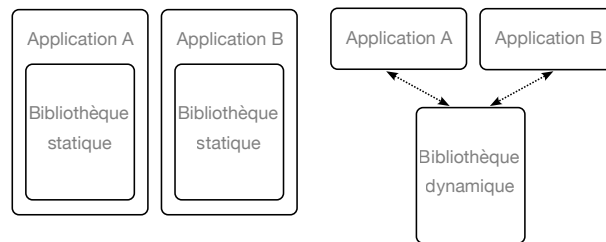


Fig. 9.7: Bibliothèques statiques et dynamiques.

9.8.4 Exécution

Un programme compilé avec une bibliothèque statique est directement exécutable puisque tout le code est le programme. Pour un programme compilé avec une bibliothèque dynamique, il faut spécifier le chemin où trouver cette bibliothèque au moment de l'exécution puisque le code n'est pas inclus dans le programme, mais dans la bibliothèque. Il est alors nécessaire d'affecter la variable `LD_LIBRARY_PATH` avant toute exécution ou bien de l'ajouter de manière permanente dans le fichier « `.profile` ».



Il est possible de lister les bibliothèques dynamiques liées à un programme en appelant la commande : « `ldd programme` ».

9.8.5 Compilation avec des bibliothèques

Si les deux versions d'une bibliothèque existent, par défaut le compilateur prend la bibliothèque dynamique (« `.so` »). Pour sélectionner la version, il faut mettre `-static` ou `-dynamic` devant chaque nom de bibliothèque. Par exemple pour une bibliothèque statique « `libname.a` » on aura :

```
$ gcc [-static] -lname
```

et pour une bibliothèque dynamique « `libname.so` » :

```
$ gcc [-dynamic] -lname
```



Attention à l'ordre dans lequel on place les bibliothèques, car la recherche s'effectue de la gauche vers la droite.

9.8.6 Construction d'une bibliothèque

a) Bibliothèque statique

Afin de construire la bibliothèque statique « `libname.a` » à partir des fichiers objets du répertoire courant, il faut lancer la commande :


```
$ ar crv libname.a *.o
```

avec les options *c* (*creation*), *r* (*replace*) et *v* (*verbose*).

Sur certains systèmes il faut ensuite utiliser la commande :

```
$ ranlib libname.a
```

La commande suivante permet de visualiser le contenu de la bibliothèque :

```
$ ar tv libname.a
```

b) Bibliothèque dynamique

La construction d'une bibliothèque dynamique revient à construire un programme à peu près normal, excepté qu'il n'y a pas de fonction `main()` et que le code doit être « relogeable » (production des « .o » avec l'option `-fPIC` *-- positioning independant code*).

Les directives de compilation dépendent des systèmes :

- Sous Linux : `gcc -shared -o libname.so *.o`
- Sous Mac OS X : `gcc -dynamiclib -o libname.dylib *.o`
- Sous Ms-Windows (MingW) : `gcc -shared -o libname.so *.o`

9.9 Vim et la compilation

Depuis le mode interactif, il suffit d'entrer la commande suivante :

```
:!gcc -Wall -Wextra -ansi -pedantic test.c -o test
```



Exercices

Exercice 9.1

Dans la section « compilation » de l'exemplier, testez les commandes suivantes :

- `gcc ex01_first.c ; ./a.out` (compilation et exécution).
- `gcc ex01_first.c -o first` (renommer l'exécutable).
- `gcc -Wall ex01_warnings.c` (compilation avec affichage des avertissements).
- `gcc -Wall -Wextra ex01_warnings.c` (compilation avec affichage des avertissements, notamment ceux sur les intervalles de types).
- `gcc -Wall -Wextra -ansi ex01_ansi_pedantic.c` (pour les commentaires //).
- `gcc -Wall -Wextra -pedantic ex01_ansi_pedantic.c` (pour l'utilisation d'une variable pour dimensionner le tableau).

Exercice 9.2

Dans la section « compilation » de l'exemplier, testez les commandes suivantes :

- `gcc -E ex02_comments.c > ex02_comments.i` (élimination des commentaires) -- visualisez le fichier produit à l'aide de la commande `more`.

- `gcc -E ex02_macros.c > ex02_macros.i` (remplacement des macros) -- visualisez le fichier produit à l'aide de la commande `less`.
- `gcc -E ex02_include.c > ex02_include.i` (inclusion de fichiers « .h »)
- `gcc -E ex02_include_more.c > ex02_include_more.i` (inclusion de fichiers « .h »).
- `gcc -Wall -Wextra -ansi -pedantic ex02_ifdef.c` (avertissement pour l'absence d'inclusion). Recommencez avec l'option `-DINCLUSION` pour mettre en évidence l'utilisation des directives `#ifdef`.

Exercice 9.3

Dans la section « compilation » de l'exemplier, testez les commandes suivantes :

- `gcc -S ex03_assembler.c` (permet de générer le code assembleur). -- visualisez le fichier résultant « `ex03_assembler.s` » à l'aide de la commande `more`.
- `gcc -c ex03_assembler.s` (appel de l'assembleur `as`).
- Puis : `od -x ex03_assembler.o > ex03_assembler.o.txt` (redirection de l'affichage en octal de la copie binaire brute) -- visualisez le fichier produit à l'aide de la commande `less`.

Exercice 9.4

Dans la section « compilation » de l'exemplier, testez les commandes suivantes :

- `gcc -S ex03_assembler.c` (permet de générer le code assembleur). -- visualisez le fichier résultant « `ex03_assembler.s` » à l'aide de la commande `more`.
- `gcc -c ex03_assembler.s` (appel de l'assembleur `as`).
- Puis : `od -x ex03_assembler.o > ex03_assembler.o.txt` (redirection de l'affichage en octal de la copie binaire brute) -- visualisez le fichier produit à l'aide de la commande `less`.

Exercice 9.5

Corrigez le code du programme « `buggy.c` » suivant jusqu'à ce que la compilation avec les options `-Wall`, `-Wextra`, `-ansi` et `-pedantic` se fasse sans erreurs ni avertissements.

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

int main(void) {
    int index;

    for (index; index < 10; i++) {
        printf("%d * %d = %p\n", index, index, square());
    }

    return EXIT_SUCCESS;
}

int square(int number) {
    return number * number;
}
```

Exercice 9.6

Soit les fonctions `bonjour()` et `bonsoir()` ci-après qui affichent respectivement « Bonjour » et « Bonsoir » autant de fois que le nombre passé en argument :

```
void bonjour(int nombre) {  
    int i;  
  
    for (i = 0; i < nombre; i++) {  
        printf("Bonjour\n");  
    }  
    return;  
}
```

```
void bonsoir(int nombre) {  
    int i;  
  
    for (i = 0; i < nombre; i++) {  
        printf("Bonsoir\n");  
    }  
    return;  
}
```

1. Éditez ces fonctions dans les fichiers respectifs « `bonjour.c` » et « `bonsoir.c` ».
2. Compilez les fichiers afin de ne générer que les fichiers objets.
3. Créez la bibliothèque statique « `libjoursoir.a` » à l'aide de la commande `ar`. Vérifiez son contenu à l'aide de cette même commande.
4. Éditez, compilez et exécutez le fichier de test de la bibliothèque « `test_joursoir.c` ». Ce fichier ne considérera que les nombres compris entre 1 et 10.

Exercice 9.7

Réitérez les opérations avec les fichiers précédents en créant cette fois-ci la bibliothèque dynamique « `libjoursoir.so` ».

10 Commande make

« Most of you are familiar with the virtues of a programmer. There are three, of course : laziness, impatience, and hubris. » Larry Wall.

Dans ce chapitre, nous expliquons comment utiliser la commande *make* afin de construire une application composée de plusieurs fichiers.

10.1 Objectif

10.1.1 Problème

Considérons un logiciel composé de plusieurs fichiers avec des interdépendances tel que celui de la figure 10.1. Pour le compiler, nous pourrions tout rassembler dans un fichier unique, puis afin d'éviter une recompilation après chaque modification, nous utiliserions un fichier de script contenant la liste de toutes les directives de compilation. Cette solution serait en fait beaucoup trop lourde et peu efficace.

Quelles sont les questions non résolues par l'approche précédente :

1. Si on modifie le fichier « `lectab.c` », que faut-il lancer comme commande pour tout recompiler ?
2. Si on modifie « `exp.h` », quels sont les seuls fichiers à recompiler ?
3. Le programme compilé est-il bien celui qui a été obtenu avec la dernière version des fichiers ? Donc, si l'on se réfère aux dates des fichiers, quels sont ceux qu'il faut recompiler ?
4. En cas d'interruption de la compilation, d'où doit-on repartir ?

La solution à ces problèmes est d'utiliser le programme GNU *make*. *make* est un moteur de production multiplateforme qui permet d'automatiser la réponse aux questions précédentes. Il compte parmi les utilitaires de programmation les plus précieux. On pourra pour des informations plus précises se reporter à l'ouvrage de Robert Mecklenburg [9].

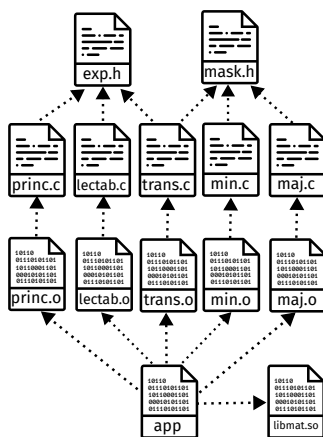


Fig. 10.1 : Exemple de logiciel composé de plusieurs fichiers avec leurs dépendances.

10.1.2 Structure

GNU *make* utilise un fichier de description nommé par défaut *Makefile* (ou *makefile*) et qui contient les directives de compilation, mais aussi les dépendances entre tous les fichiers.

10.2 Structure d'un fichier Makefile

Le fichier *Makefile* est plus qu'un simple fichier de commandes de compilation :

- il prend en compte la date de modification des fichiers pour ne pas tout reconstruire à chaque fois;
- il peut être utilisé pour n'importe quelles autres tâches que la compilation (par exemple : l'archivage avec *tar*).

10.2.1 Éléments d'un fichier Makefile

- Commentaires : ils sont monolignes et sont introduits par le caractère « # ».
- Coupure d'une ligne pour une meilleure visibilité : le caractère « \ » annule la fin de ligne (« à la C »).

10.2.2 Structure d'une règle

```
cible: dépendance1 ... dépendancei
    commande1
    # ....
    commandei
```

10.2.3 Exemple

```
app: princ.o lectab.o trans.o min.o maj.o
    gcc princ.o lectab.o trans.o min.o maj.o -o app -lmatrice
```

Construction des fichiers objets.

```
princ.o: princ.c exp.h
    gcc -Wall -Wextra -ansi -pedantic princ.c -c
```

```
lectab.o: lectab.c exp.h
    gcc -Wall -Wextra -ansi -pedantic lectab.c -c
```

```
trans.o: trans.c exp.h masque.h
    gcc -Wall -Wextra -ansi -pedantic trans.c -c
```

```
min.o: min.c masque.h
    gcc -Wall -Wextra -ansi -pedantic min.c -c
```

```
maj.o: maj.c masque.h
    gcc -Wall -Wextra -ansi -pedantic maj.c -c
```

10.3 Commande make

10.3.1 Exécution

```
$ make [options] [cible]*
```



Si la cible n'est pas indiquée, *make* prend uniquement la première cible.

Dans le cas où le fichier de commande porte le nom « Makefile » ou « makefile », ce qui est préférable, alors l'appel se fait plus simplement :

```
$ make
```

Si par contre le fichier de commande porte un autre nom, ou qu'il se trouve dans un autre dossier, alors l'appel se fait de la manière suivante :

```
$ make -f fichier
```

10.3.2 Algorithme de *make*

```
POUR chaque cible choisie ou la première cible FAIRE
    lancer make récursivement sur chaque dépendance de la cible
    SI make retourne une erreur
        ALORS arrêter make
    FINSI
    SI la cible est plus ancienne qu'une des dépendances
        ou la cible n'existe pas
        ALORS lancer les commandes associées à la cible
    SINON écrire : "fichier à jour"
    FINSI
FINPOUR
```



Deux appels consécutifs à *make* ne doivent exécuter qu'une seule fois l'opération. Il peut arriver que l'on soit obligé de forcer la recompilation d'une cible. Il y a alors deux possibilités : supprimer le fichier cible ou changer la date d'un des fichiers de dépendance par l'appel « touch fichier ». On préférera le changement de date, la suppression (rm fichier) étant considérée comme dangereuse.

10.4 Cibles

Une **cible** est le nom d'un objet à réaliser. La plupart du temps, il s'agira du nom d'un fichier à construire (y compris son chemin absolu ou relatif).

Exemples : « executable », « ../bin/executable », « fft.o », « ../objet/fft.o ».

Il est possible de mettre plusieurs cibles pour une seule règle :

`fft.o matrix.o topology.o`:

Cela signifie que pour construire la cible « `fft.o` », « `matrix.o` » ou « `topology.o` », il faut utiliser cette règle.

10.5 Dépendance

Une **dépendance** est un fichier dont dépend la cible pour être réalisée. Le nom d'une dépendance comporte le chemin absolu ou relatif si nécessaire.



Une cible A dépend du fichier B si la modification de B entraîne celle de A.

Exemple :


```
matrix.o : matrix.c matrix.h
```

Dans cet exemple, la réalisation du fichier « `matrix.o` » dépend des fichiers « `matrix.c` » et « `matrix.h` ».



En général les dépendances sont les fichiers en-têtes, les fichiers objets ou encore des bibliothèques. Toutefois, on ne met que les fichiers maintenus par le projet et non pas tous les fichiers inclus. Par exemple, on ne mettra pas « `stdio.h` ».

10.6 Commande dans une règle

Une **commande** est introduite pour une règle **obligatoirement** par une tabulation . La plupart du temps, ce sont des commandes *shell* dont le but est de réaliser la cible. Il peut y avoir plusieurs commandes pour une même cible, par exemple :

```
battleship : main.o game.o graphics.o input.o
    gcc main.o game.o graphics.o input.o -o battleship
    echo "C'est fini !"
```

Le *shell* `/bin/sh` est utilisé quelque soit l'interpréteur de l'utilisateur. Il est possible de modifier le *shell* utilisé par défaut en redéfinissant la macro `SHELL` dans le fichier `Makefile`.

Les commandes s'exécutent en séquentiel par appels successifs à l'interpréteur de commandes. Chaque exécution est donc indépendante des autres (pas de passage de variable d'environnement par exemple). Il n'est donc pas possible de changer de dossier par un appel à « `cd` » si les commandes suivantes ne sont pas dans le même *shell*. Il faut donc utiliser « `commande1; commande2; ...` » pour que cela s'exécute dans le même *shell*. D'où la première version est incorrecte, la seconde est correcte :

```
# Incorrect
battleship : main.o game.o graphics.o input.o
    cd dir;
    gcc main.o game.o graphics.o input.o -o battleship

# Correct
battleship : main.o game.o graphics.o input.o
    cd dir ; gcc main.o game.o graphics.o input.o -o battleship
```

Parmi les commandes, il est possible d'appeler récursivement le programme *make* sur une autre cible par :

```
autres :
    make cible
```

10.7 Règles explicites

Les **règles explicites** sont des règles avec des noms de cible explicites : fichier ou identificateur.

10.7.1 Cibles fichier

Ces **cibles** sont des fichiers à réaliser. Un fichier peut être précédé du nom du dossier le contenant.

Exemple :

```
prog : princ.o trans.o lectab.o min.o maj.o
    gcc -o prog princ.o lectab.o trans.o min.o maj.o -lc
    echo "programme compilé"

princ.o : exp.h princ.c
    gcc -c princ.o

princ.c :
    emacs princ.c
```

10.7.2 Cibles non-fichiers

Il est possible de définir des cibles qui ne correspondent pas à des fichiers : ce sont des identificateurs.

Par exemple et par pure convention :

- La cible par défaut « **all** » est toujours placée en première position.
- La cible de nettoyage « **clean** » permet de supprimer les fichiers intermédiaires devenus inutiles tels que les fichiers objets (« **.o** »).
- La cible de post-installation « **distclean** » qui permet de revenir dans la configuration d'avant le premier appel à *make*.

Exemple :

```
all : executable clean
```

10.7.3 Cibles modules de bibliothèque (non présenté)

La référence à un module d'une bibliothèque s'effectue en plaçant le nom de ce module entre parenthèses derrière le nom de la bibliothèque. Par exemple, pour le module `trans.o` de la bibliothèque `bibl.a`, on écrira : « `bibl.a(trans.o)` ».

```
bibl.a : trans.o lectab.o min.o maj.o
    ar r trans.o lectab.o min.o maj.o

bibl.a(trans.o) : trans.o
    ar r bibl.o trans.o
```


10.8 Macros

10.8.1 Macros internes

Une **macro** est une variable dont la valeur est une chaîne de caractères. Par convention, le nom des macros est en majuscules. La macro est remplacée par sa valeur chaque fois qu'elle apparaît sous forme de référence.

Pour définir une macro, on écrira :

```
MACRO=chaîne
```

Pour y faire référence, on utilisera `$(MACRO)`.



Toute macro non définie a comme valeur la chaîne vide « "" ».

Exemple d'utilisation d'une macro :

```
MODULE = trans.o lect.o min.o maj.o
```

```
prog : princ.o $(MODULE)
      ld -o prog $(MODULE) -lm
```

Enfin, il est possible de réaliser une concaténation sur la valeur d'une macro de la manière suivante : `MODULE+=max.o`.



Les macros sont remplacées récursivement jusqu'à épuisement (`A=$B` et `B="a"` implique que `A="a"`).

10.8.2 Macros internes classiques

Les **macros internes** sont définies dans le tableau suivant :

Macro interne	Description
<code>\$(MAKE)</code>	Nom du programme <i>make</i> .
<code>\$(MAKEFLAGS)</code>	Liste des options de l'appel de <i>make</i> .
<code>\$(SHELL)</code>	Nom de l'interpréteur de commande courant.
<code>\$(CC)</code>	Commande pour le compilateur C (par défaut <code>cc</code>).
<code>\$(CFLAGS)</code>	Liste des options du compilateur C.
<code>\$(LDFLAGS)</code>	Liste des options de l'éditeur de liens C.
<code>\$(CPPFLAGS)</code>	Liste des options du précompilateur.
<code>\$(CXX)</code>	Commande pour le compilateur C++ (par défaut <code>CC</code>).
<code>\$(HOST_ARCH)</code>	Type de l'architecture de la machine.

Exemple d'une cible avec macros :

```
../objet/graphics.o : graphics.c
    $(CC) -c graphics.c $(CPPFLAGS) $(CFLAGS) -o ../objet/graphics.o
```

L'accès aux variables du *shell* oblige à mettre `$(...)` :

```
echo $(PATH)
echo $(LD_LIBRARY_PATH)
```

10.8.3 Macros internes spéciales

Les macros prédéfinies sont regroupées dans le tableau suivant :

Macro prédéfinie	Description
<code>\$@</code>	Contient le nom de la cible -- si la cible contient plusieurs fichiers, alors elle contient celle qui a déclenché la règle.
<code>\$*</code>	Contient le nom de la cible sans suffixe.
<code>^</code>	Nom de toutes les dépendances séparées par des espaces.
<code>\$?</code>	Contient la liste des dépendances plus récentes que la cible.
<code>\$<</code>	Nom de la première dépendance de la liste.
<code>\$(nom.o)</code>	Cette macro est évaluée quand le nom de la ligne de dépendance est celui d'un membre d'une bibliothèque. Dans ce cas le nom est de la forme « <code>bibl(nom.o)</code> ». La macro <code>\$@</code> a pour valeur « <code>bibl</code> » et la macro <code>\$(nom.o)</code> vaut « <code>nom.o</code> ».

Exemple : afficher la dépendance qui déclenche la règle.

```
princ.o : princ.c exp.h
    @echo $?
    @gcc $*.c -o $@
```

10.9 Règles implicites

Les **règles implicites** permettent d'écrire une seule règle pour un ensemble de règles explicites très semblables. Ce sont des règles qui s'appliquent à des types de fichiers (par exemple par leur suffixe) et non à des fichiers nommés.

10.9.1 Définition d'une règle implicite

Elle se fait par les suffixes :

```
# Règle traditionnelle
.c.o :
    $(CC) $(CPPFLAGS) $(CFLAGS) -c $< -o $@

# Conversion d'un fichier jpeg en gif
.jpg.gif :
    convert $< $@
```

10.9.2 Règles implicites prédéfinies

`make` contient déjà un ensemble de règles implicites. Par exemple, pour compiler un fichier « `.c` » en fichier « `.o` » :

```
$ make CC=gcc CPFLAGS="-O3 -g" -LDFLAGS=-lm matrix.o
```

La liste des préfixes qui apparaissent dans les règles implicites sont définis par la directive `.SUFFIXE`. Les règles implicites sont examinées dans l'ordre précisé par cette directive.

Il est possible d'ajouter de nouveaux suffixes ou de désactiver les règles implicites associées à un suffixe.

- `.SUFFIXE` : sans contenu réinitialise la liste.
- `.SUFFIXE` : `<suffixe>*` permet d'ajouter de nouveaux suffixes.

10.9.3 Règle sans commandes associées

Il peut y voir plusieurs fois la déclaration de dépendance pour une même cible. Il y a concaténation des dépendances, mais une seule liste de commandes.

La règle suivante :

```
obj/%.o : src/%.c include/%.h
```

permet de séparer la définition des dépendances des commandes.

10.9.4 Définition d'une règle implicite spécifique GNU

Même syntaxe que les règles explicites, simplement la cible contient le caractère `%` (et un seul!) qui remplace n'importe quelle suite de caractères. Par exemple `%.c` est un motif qui peut être associé à n'importe quel fichier C du dossier. Une fois le fichier sélectionné, le caractère `%` est remplacé par sa valeur partout où il est utilisé dans la définition de la règle. Attention : le métacaractère n'est pas valable dans les lignes de commandes (il faut utiliser `$@`, `$^`, etc.).

```
obj/%.o : src/%.c include/%.h # ajout des fichier entêtes
$(CC) $(CFLAGS) $(CPPFLAGS) -c $< -o $@
```

```
bin/% : obj/%.o $(LIBS)
$(CC) $(CFLAGS) $(LIBS) $< -o $@
```

10.9.5 Exemple d'un Makefile plus complet

```
.PHONY : clean
OBJECTS = princ.o lectab.o trans.o min.o maj.o

princ.o : exp.h masque.h
lectab.o : exp.h
trans.o : exp.h
min.o : masque.h
maj.o : masque.h

.SUFFIXES: .c.o
```

```
.c.o :
    $(CC) $(CFLAGS) $<

princ :$(OBJECTS)
    $(CC) $(LD_FLAGS) -o princ $(OBJECTS)

clean :
    $(RM) $(OBJECTS) princ *~

distclean : clean
    $(RM) princ
```

10.10 Directives

10.10.1 Inclusion de fichiers

`include <fichier>` correspond à une inclusion d'un fichier sans interprétation (« à la C »). Le fichier peut être donné avec son chemin en absolu ou en relatif. Il peut aussi être localisé grâce à l'option `-I` du *make*.

Intérêts :

1. Centraliser les définitions partagées à tous les fichiers *Makefile*.
2. Faire un fichier uniquement avec les cibles et leurs dépendances (« `gcc -M` »). Il ne reste plus qu'à faire `include <fichier2>` dans le fichier *Makefile*.

Option du préprocesseur de GNU *gcc* :

```
-MM  Génère les dépendances directes :
      « fft.o: fft.c fft.h »
      uniquement au premier niveau des #include " ".
-M   Génère toutes les dépendances d'un fichier C (les fichiers
      « .h ») récursivement.
```

Exemple : `gcc -I../include -MM fft.c > fichier.dependencies`

10.10.2 Directive par défaut

- `.DEFAULT` est la règle par défaut en cas d'erreur.

Par exemple :

```
.DEFAULT : echo "$< n'existe pas."
```

10.10.3 Directives qui caractérisent l'exécution

make affiche les commandes avant de les exécuter et s'arrête dès qu'une commande produit une erreur ou qu'elle est interrompue (`Ctrl` + `C`). Les directives suivantes sont donc très utiles :

- **.IGNORE** : l'échec d'une commande arrête *make*. Cette directive permet d'ignorer les erreurs et ne s'arrête pas après la première erreur. Pour ignorer l'échec d'une commande seule, il suffit de mettre un tiret - devant celle-ci : « `-rm matrix.o` ».

clean :

```
-rm *~
-rm *.o # non effectué s'il n'y a pas de fichier *~
```

- **.SILENT** : *make* affiche les commandes avant de les exécuter. Cette directive rend l'exécution silencieuse. Pour empêcher l'affichage d'une commande en particulier on place un caractère @ devant celle-ci :

prog : a.o b.o

```
@echo "Coucou"
cc a.o b.o -o prog
```

- **.PRECIOUS** : permet d'indiquer les fichiers à ne pas effacer en cas d'échec (par exemple les fichiers intermédiaires). En effet, lorsque *make* est interrompu ou en échec il supprime tous les fichiers en cours ainsi que les fichiers intermédiaires. Pour l'en empêcher on indique après le caractère « : » la liste des fichiers à conserver.
- **.PHONY** : liste des cibles qui ne sont pas des fichiers. Une cible non-fichier est toujours exécutée. Par contre, si un fichier porte malencontreusement le même nom qu'une cible alors la cible ne sera plus exécutée. **.PHONY** évite ce problème en considérant cette cible indépendante de tout fichier. Toute cible non fichier doit donc se trouver dans **.PHONY**.

.PHONY : clean

clean :

```
rm -f *.o a.out
```

10.10.4 Commande wildcard (spécifique GNU)

La commande **wildcard** permet de récupérer des noms de fichiers de manière dynamique.

```
SRC=$(wildcard *.c) # récupère tous les fichiers d'extension "c".
```

```
OBJECT=$(SRC:.c:.o)
```

À utiliser avec parcimonie puisque le développeur n'a plus la maîtrise des dépendances.

10.11 Options de la commande make

```
$ make [option]* [definition macro]* [cibles]*
```

10.11.1 Options

Ce fichier de commande porte le nom « Makefile » ou « makefile », *make* peut être appelé sans argument.

- **make -f <file>** : exécution de *make* avec un fichier de commandes (ex. : `make -f Makefile.win`).
- **make -s** : mode silence (équivalent à **.SILENT**).
- **make -k** : continue même après une erreur (équivalent à **.IGNORE**).
- **make -Ifolder** : le nom d'un dossier pour les directives *include*.

10.11.2 Déboguer un Makefile

- `make -d` : affiche les dépendances quand on recompile. (`--debug <option>` (`basic`, `all`)).
- `make -n` (ou `--just-print`) : mode passif, affiche les commandes qu'il exécuterait sans l'option `-n`.
- `make -p` (ou `--print-data-base`) : liste des macros du fichier « `Makefile` ».
- `make --warn-undefined-variables`.

10.11.3 Définition de macros

```
$ make [<macro=valeur>]* [<cibles>]*
```

Exemple d'appel :

```
$ make CC="gcc" LDFLAGS="-lm"
```

On peut ainsi redéfinir des macros. Par contre celles du fichier sont prioritaires.

10.12 Makefile et Vim

La commande « `:make` » permet de lancer l'outil en supposant la présence du fichier « `Makefile` » dans le dossier courant.



Tout projet doit donc comporter un fichier « `Makefile` » !

Deux exécutions consécutives de `make` doivent afficher le message :

```
make: `xxxxxx' is up to date.
```

10.13 Outils de création de makefile

De nombreux environnements de développement intégré (IDE -- *integrated development environment*) permettent d'occulter la création des fichiers « `Makefile` ». Par exemple [Visual Studio](#), [CLion](#), etc.

La difficulté de gérer les gros projets a permis l'apparition de certains outils censés améliorer la création des fichiers « `Makefile` » :

- `CMake` : le plus utilisé (utilise un langage spécialisé);
- `QMake` (Qt), `NMake` (Ms-Windows), etc.;
- `automake`, `autoconf` : très complexes.

Ces outils présentent plusieurs avantages :

- adaptation au système d'exploitation;
- vérification de la configuration courante;
- paramétrage de la configuration en fonction des ressources présentes.

10.13.1 *make* : l'ancêtre

Principe : un fichier « *Makefile* » décrit les commandes à exécuter.

Avantages :

- outil standard;
- principe de fonctionnement simple.

Inconvénients :

- syntaxe des commandes (*shell*);
- portabilité des commandes.

10.13.2 *autoconf* / *automake* : GNU

Principe : *automake* génère des squelettes de *Makefile* (« *Makefile.in* » à partir d'un fichier « *Makefile.am* »). Le programme *autoconf* génère un script de création de fichiers *Makefile* (nommé *configure*) à partir de fichiers « *configure.ac* » et « *Makefile.in* ».

Avantages :

- très répandu;
- support pour l'exécution de suites de tests.

Inconvénients :

- apprentissage difficile;
- une syntaxe pour *autoconf*, une pour *automake*.

10.13.3 *QMake* / *TMake* : Trolltech (pour Qt et KDE)

Principe : génère un fichier *Makefile* à partir d'un fichier « *.pro* ».

Avantages :

- apprentissage simple;
- utilitaire de génération de fichiers « *.pro* » (*progen*).

Inconvénients :

- pas de support des suites de tests.
- pas très puissant en dehors de Qt.

10.13.4 *CMake* : Kitware (VTK)

Principe : génère un fichier *Makefile* (Unix) ou un projet Visual Studio (Ms-Windows) à partir d'un fichier « *CmakeLists.txt* ». C'est l'outil de base proposé pour les projets sous CLion (bin qu'il gère aussi les fichiers « *Makefile* »).

Avantages :

- actuellement à la mode;
- syntaxe relativement simple.

Inconvénients :

- chemins absolus dans le(s) Makefile(s) généré(s). Il est nécessaire de réexécuter *cmake* si on déplace le dossier.

10.13.5 Ant (Apache pour Java)

Principe : le fichier Makefile est remplacé par un fichier « `build.xml` » écrit en XML.

Avantages :

- presque un standard;
- complet pour Java.

Inconvénients :

- syntaxe du fichier en XML;
- limité presque à Java.



L'outil *Maven* tend à remplacer *Ant*.

10.13.6 Gradle (Android pour Java)

Principe : le Makefile est remplacé par un fichier « `build.gradle` » écrit en langage Groovy.

10.14 Exemple complet

Voici un exemple de fichier Makefile utilisant des règles implicites :

```
CFLAGS=-Wall -Wextra -ansi -pedantic -I./include -O2

.PHONY: all distclean clean tar

all: bin/app

bin/app: object/princ.o object/lectab.o object/trans.o object/min.o
object/maj.o
    gcc $^ $(LD_FLAGS) -o $@ -lmatrice

# Construction des fichiers objets.
object/%.o:
    gcc $(CPP_FLAGS) $(CFLAGS) $< -c -o $@

object/princ.o: src/princ.c include/exp.h
object/lectab.o: src/lectab.c include/exp.h
object/trans.o: src/trans.c include/exp.h include/masque.h
object/min.o: src/min.c include/masque.h
object/maj.o: src/maj.c include/masque.h
```



```
clean:
    rm -f object/*.o
    rm -f include/*~
    rm -f src/*~
    rm -f *~

distclean: clean
    rm -f bin/app

tar: distclean
    tar cvfz exemple.tgz .
```

Exercices

Exercice 10.1

Écrire le fichier `Makefile` correspondant à la situation suivante et permettant de produire le programme `exo1` :

- Le fichier « `exo1.c` » contient la fonction `main()` et inclut les fichiers suivants : « `stdio.h` » et « `erreur.h` ».
- Le fichier « `erreur.c` » définit la fonction `erreur()` dont le prototype se trouve dans le fichier « `erreur.h` » et qui est appelée depuis « `exo1.c` ».

Exercice 10.2

Écrire le fichier `Makefile` correspondant à la situation suivante et permettant de produire le programme `qmotor` :

- Le fichier « `gmotor.c` » contient la fonction `main()` et inclut les fichiers suivants dont il utilise des fonctions : « `stdio.h` », « `string.h` », « `math.h` », « `common.h` », « `util.h` » et « `motor.h` ».
- Le fichier « `util.c` » inclut les fichiers suivants : « `stdio.h` », « `common.h` » et « `util.h` ».
- Le fichier « `common.c` » inclut les fichiers suivants : « `stdio.h` » et « `common.h` ».
- Le fichier « `libmotor.a` » contient les fonctions déclarées dans le fichier « `motor.h` ».

Exercice 10.3

Écrire le fichier `Makefile` permettant de générer le programme de jeu *worms* basé sur la bibliothèque *ncurses* (bibliothèque libre fournissant une API pour le développement d'environnements en mode texte) et respectant le diagramme de dépendances suivant :

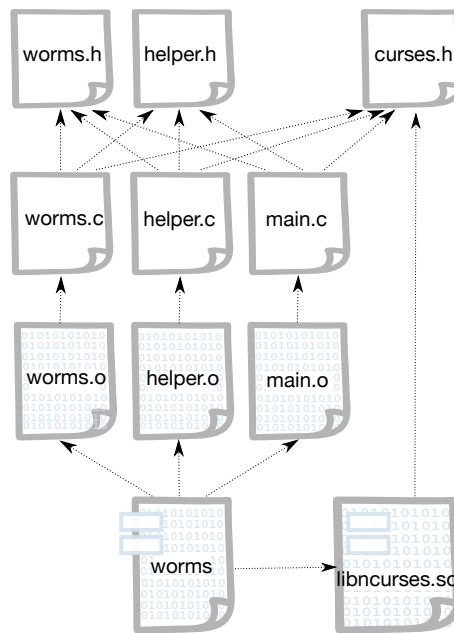


Fig. 10.2 : Diagramme de dépendances du jeu worms.

11 Mise au point du logiciel

« How to test? is a question that cannot be answered in general. When to test? however, does have a general answer : as early and as often as possible. » Bjarne Stroustrup.

11.1 Introduction

La mise au point d'un logiciel passe par deux grandes étapes : la détection des erreurs et l'optimisation du code.

On distingue quatre types d'erreurs :

1. Les **anomalies** : le programme a un comportement aléatoire.
2. Les **erreurs d'exécution** : le programme s'arrête de façon imprévue.
3. Les **fuites de mémoire** : les erreurs liées à l'utilisation de la mémoire.
4. Les **erreurs d'algorithme** : le programme ne fait pas ce qu'il faut.

En ce qui concerne l'optimisation du code, il s'agit de déterminer les parties du code à optimiser. Il ne sert à rien de tout optimiser!

11.2 Détection des anomalies

11.2.1 Notion d'anomalie

Une anomalie n'est pas une erreur du langage, mais une mauvaise utilisation de celui-ci. De ce fait, un programme exécutable est quand même généré par le compilateur. Les anomalies confèrent un comportement aléatoire au programme et son code devient non portable puisqu'il ne respecte pas la norme. De ce fait, chaque système le traite différemment et produit un comportement différent.

Une anomalie doit être généralement considérée comme une erreur. Il est donc nécessaire d'éliminer toutes les anomalies.

11.2.2 Types d'anomalies

Voici quelques types d'anomalies fréquemment rencontrés :

- incohérence des appels de fonctions : entre les paramètres effectifs et réels et la valeur de retour;
- variables non utilisées (occupation de la mémoire inutile);
- parties de programmes inutiles ou inaccessibles (occupation de la mémoire inutile);
- variables utilisées, mais non définies (pas d'initialisation à 0 par défaut);
- variables définies deux fois de suite;
- détection des arguments de fonctions non utilisées et amenant à un comportement aléatoire;
- détection des problèmes de portabilité.

Afin de détecter les anomalies, il est possible d'utiliser les outils qui suivent.

Compilateur

Il faut penser à utiliser les options `-Wall` et `-Wextra` afin de générer des avertissements. De plus, il est possible de mettre l'option `-Werror` lors de la compilation si l'on veut transformer des anomalies en erreurs.

Commande splint

`splint` permet une analyse de type `-Wall` de manière plus poussée. Par exemple, `splint` détecte même les appels de `printf()` qui ont été réalisés sans utilisation de la valeur de retour. En effet, il faudrait écrire en toute rigueur : `(void)printf(...)`.

11.3 Détection des erreurs d'exécution

Il existe trois types d'erreurs d'exécution, mais n causes possibles. Une erreur d'exécution génère l'un des trois messages suivants :

- *segmentation fault* (avec copie de la mémoire `--coredump`);
- *bus error* (avec copie de la mémoire);
- *floating point exception* (division par zéro, NaN pour *Not a Number*).

11.3.1 Fichier « core »

Après chaque erreur suivie d'une copie de la mémoire, un fichier nommé « core » est généré. C'est une « image » de la mémoire du programme au moment de l'erreur. Cette méthode est utilisée pour l'analyse *a posteriori* (ex. : message sous Ms-Windows lors d'une erreur).

L'analyse du contenu du fichier « core » oblige à savoir « parler le binaire », sauf si le programme a été compilé avec `-g` auquel cas, le code est analysable avec un programme de déverminage.

Pour activer la construction de fichiers « core », il faut ajouter la ligne qui suit dans le fichier « `.profile` » :

```
ulimit -C unlimited
```

Afin d'annuler la production du fichier « core », on écrira :

```
ulimit -C 0
```

11.3.2 Outil : le dévermineur

`gdb` est un **dévermineur** (ou débogueur) symbolique de programmes C en mode textuel, c'est-à-dire un utilitaire d'aide à la mise au point. On trouvera dans l'ouvrage de Norman Matloff tous les compléments nécessaires [7].



Le programme `ddd` propose une surcouche graphique au dévermineur GNU `gdb`.

Afin d'utiliser le dévermineur, il faut d'abord compiler les programmes avec l'option `-g`. Cette option est indispensable pour que le dévermineur puisse exécuter le programme en mode pas à pas en reliant l'exécutable au code source.

Commandes de contrôle de l'exécution

- `file [fichier]`, `exec-file [fichier]` : charge en mémoire le programme à exécuter.
- `run [arguments]`, `r [arguments]` : lancement du programme jusqu'à sa fin normale, une condition d'arrêt ou encore une erreur fatale. Les arguments de la ligne de commande s'ils existent doivent être indiqués à la suite de `run`.
- `continue` : reprend l'exécution du programme après un point d'arrêt, jusqu'à sa fin normale, un nouveau point d'arrêt ou une erreur fatale.
- `step [nombre]`, `s [nombre]` : exécute une (ou un nombre d') instruction(s) du programme.
- `next, n` : exécute une instruction, mais traite l'appel d'une fonction comme une instruction unique.

Commandes relatives aux points d'arrêt

- `break [fonction] | [numéro de ligne]` : définit un point d'arrêt du programme dès l'entrée dans la fonction ou à l'atteinte du numéro de ligne du code source. Par exemple, `break main` permet de stopper le programme dès son lancement.
- `clear [fonction] | [numéro de ligne]` : supprime les points d'arrêt attachés à la fonction ou au numéro de ligne indiqué.

Commandes de traçage

- `print [variable]`, `p[variable]` : affiche la valeur de la variable.
- `display [variable]` : affiche la valeur de la variable après chaque `break`.
- `set variable=value` : assigne une valeur à une variable.
- `where` : indique la ligne sur laquelle s'est arrêté le programme, soit sur une erreur fatale soit à un point d'arrêt, soit encore après une interruption volontaire à l'aide de `Ctrl` + `c`.
- `up` : remonte dans la pile d'appels des fonctions (les seules variables visibles sont celles de la fonction en haut de la pile d'appels).
- `down` : redescend dans la pile d'appels des fonctions.

`gdb` et `Vim`

Nous avons vu qu'il était possible sans quitter `Vim` de compiler un fichier, d'exécuter un programme, voire de lancer n'importe quelle commande Unix. De la même manière, il est possible d'exécuter `gdb` et de l'associer au fichier en cours à l'aide du greffon `TermDebug`. On installera et activera celui-ci à l'aide des commandes suivantes :

```
:packadd termdebug
:Termdebug
```

`Vim` découpe alors la fenêtre en trois panneaux dont, le panneau de contrôle du débogueur, celui de la sortie d'exécution du programme et celui d'édition comme le montre la figure 11.1.

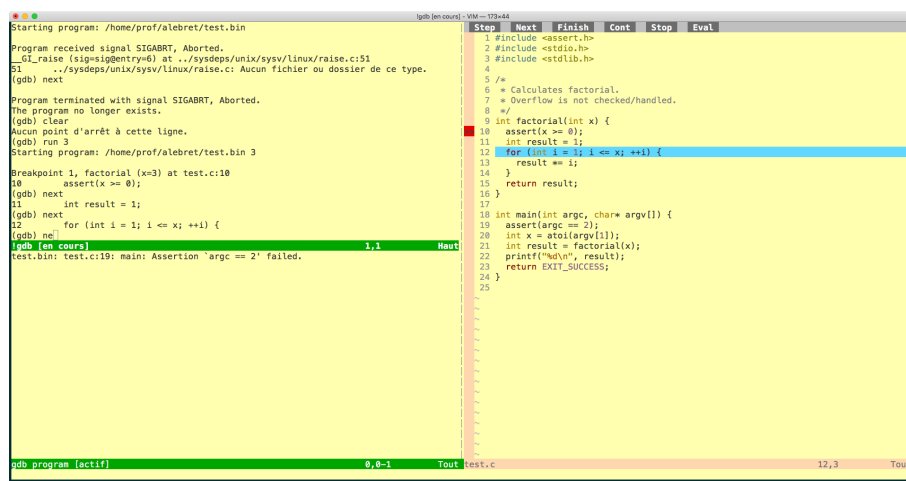


Fig. 11.1 : Vim en mode déverminage avec GNU gdb.



Notre conseil : toujours compiler les programmes avec l'option `-g` et ne la supprimer que lors de la génération de la version « commerciale ».

11.4 Détection des fuites de mémoire

L'outil **valgrind** permet d'espionner la gestion de la mémoire au cours de l'exécution du programme. Il permet entre autres de mettre en évidence les problèmes de libération ou de débordement de la mémoire. Toutefois, Il ne permet de détecter les mauvaises utilisations de la mémoire que pour l'exemple testé, c'est-à-dire qu'il ne parcourt pas toutes les branches d'un programme.

L'outil prend comme argument le nom du programme exécutable :

```
$ valgrind a.out
```

Pour avoir les noms des symboles du programme, il faut préalablement avoir compilé avec l'option `-g`.

Exemple

Soit le fichier « prog.c » suivant :

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

int main(int argc, char **argv) {
    char input_s[20] = "A new message";
    char *output_s = NULL;

    strcpy(output_s, input_s);
    puts(output_s);

    return EXIT_SUCCESS;
}
```

Ce code compile parfaitement, mais l'exécution du programme provoque une erreur de segmentation.

En lançant *gdb* :

```
jsaigne@cybele:~$ gdb prog
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
...
(gdb) break 6
Breakpoint 1 at 0x4005f4: file prog.c, line 6.
(gdb) run
Starting program: /home/prof/jsaigne/prog

Breakpoint 1, main (argc=1, argv=0x7ffffffe9c8) at prog.c:6
6      char input_s[20] = "A new message";
(gdb) next
7      char *output_s = NULL;
(gdb) next
9      strcpy(output_s, input_s);
(gdb) next

Program received signal SIGSEGV, Segmentation fault.
__strcpy_sse2_unaligned ()
    at ../sysdeps/x86_64/multiarch/strcpy-sse2-unaligned.S:730
(gdb) next

Program terminated with signal SIGSEGV, Segmentation fault.
The program no longer exists.
(gdb) quit
jsaigne@cybele:~$
```

Nous décidons d'utiliser la commande *valgrind* qui retourne alors les informations suivantes :

```
jsaigne@cybele:~$ valgrind ./prog
==26823== Memcheck, a memory error detector
==26823== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==26823== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==26823== Command: ./prog
==26823==
==26823== Invalid write of size 1
==26823==    at 0x4C3106F: strcpy (vg_replace_strmem.c:506)
==26823==    by 0x4006D9: main (prog.c:10)
==26823== Address 0x520404d is 0 bytes after a block of size 13 alloc'd
==26823==    at 0x4C2DB8F: malloc (vg_replace_malloc.c:299)
==26823==    by 0x4006C2: main (prog.c:9)
==26823==
==26823== Invalid read of size 1
==26823==    at 0x4C30F74: strlen (vg_replace_strmem.c:454)
==26823==    by 0x4EA969B: puts (ioputs.c:35)
==26823==    by 0x4006E5: main (prog.c:11)
==26823== Address 0x520404d is 0 bytes after a block of size 13 alloc'd
==26823==    at 0x4C2DB8F: malloc (vg_replace_malloc.c:299)
==26823==    by 0x4006C2: main (prog.c:9)
==26823==
A new message
==26823==
```



```

==26823== HEAP SUMMARY:
==26823==      in use at exit: 13 bytes in 1 blocks
==26823==    total heap usage: 2 allocs, 1 frees, 1,037 bytes allocated
==26823==
==26823== LEAK SUMMARY:
==26823==    definitely lost: 13 bytes in 1 blocks
==26823==    indirectly lost: 0 bytes in 0 blocks
==26823==    possibly lost: 0 bytes in 0 blocks
==26823==    still reachable: 0 bytes in 0 blocks
==26823==           suppressed: 0 bytes in 0 blocks
==26823== Rerun with --leak-check=full to see details of leaked memory
==26823==
==26823== For counts of detected and suppressed errors, rerun with: -v
==26823== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)

```

11.5 Détection des erreurs d'algorithme

Lorsque le programme ne donne pas les résultats attendus, on se retrouve à devoir détecter et corriger les erreurs d'algorithme. Deux solutions s'offrent à nous :

- utiliser un développement itératif et incrémental.
- définir des tests à chaque étape.

Plusieurs outils ou techniques complémentaires sont utilisables pour cela.

11.5.1 Gdb comme un interpréteur de code (++++)

GNU *gdb* permet de tracer le programme pas à pas et d'afficher la valeur des variables.

11.5.2 Méthode des traces (++)

La méthode des traces consiste à parsemer le code d'appels à `printf()` et `getchar()` afin d'afficher des valeurs de calculs intermédiaires.

```

#ifdef DEBUG
    fprintf(stderr, "Valeur=%d\n", valeur);
#endif

```

- Toujours mettre un « `\n` » après l'affichage d'une trace de manière à forcer son affichage immédiat, sinon l'affichage attend que le tampon soit plein et le programme peut alors se planter avant l'affichage, ce qui peut induire en erreur sur l'endroit du plantage.
- Il faut laisser ces traces dans le code, encadrées par des directives « `ifdef DEBUG` » ce qui permet de les réactiver à tout moment et évite le travail de suppression.
- Les traces à conserver dans le code sont utilisées pour la mise au point de l'algorithme et non pas pour le débogage qui lui utilise la fonction `assert()` et le débogueur GNU *gdb*.

11.5.3 Supprimer des parties de code (+)

Mettre en commentaire une partie du code pour cerner ce qui est la cause de l'erreur.

Depuis *Vim* :

1. Passer en mode visuel `Ctrl` + `V`.
2. Sélectionnez les lignes à commenter en appuyant de manière successive sur la touche `j` ou sur `↓`.
3. Appuyer sur la touche `I` pour passer en mode insertion.
4. Appuyer sur `//`.
5. Appuyer ensuite successivement deux fois sur la touche `Esc`.

Pour décommenter :

1. Passer en mode visuel `Ctrl` + `V`.
2. Sélectionner le bloc à décommenter en deux temps : `↓` pour sélectionner les premiers / sur les différentes lignes et `→` pour sélectionner la seconde colonne de /.
3. Appuyer sur `d`.



Le commentaire monoligne n'étant pas ANSI, il est conseillé d'utiliser `/*` en début de bloc et `*/` en fin.

11.5.4 Fonction `assert()` (++++)

`assert(test)` : si le test est faux, arrête le programme et affiche un message d'erreur avec le numéro de la ligne concernée. Si le test est vrai ne fait rien.

Exemple d'utilisation de `assert()` :

```
# Fichier exemple_assert.c
#include <assert.h>
```

```
float inversion(float n) {
    assert(n != 0);
    return 1.0F/n;
}
```

```
int main(void) {
    float x;
    x = inversion(0.0F);
}
```

La sortie est la suivante :

```
$ gcc -g exemple_assert.c -o exemple_assert
$ ./exemple_assert
.out: exemple_assert.c:5:fct: Assertion 'n!=0' failed.
abandon
```

L'appel à la fonction `assert()` est très utile (voire indispensable) à la mise au point d'un programme, surtout si le projet est développé par des personnes indépendantes.

L'appel sera utilisé pour :

- vérifier les paramètres formels d'entrée des fonctions afin d'éviter les utilisations frauduleuses de fonction et de mieux localiser les erreurs d'algorithme ici elles sont antérieures à l'appel de la fonction) -- très utile lors de la phase d'intégration;
- se prémunir contre le vieillissement d'un code source -- `assert()` ajoute des garde-fous pour aider les développeurs à réutiliser un code ou le modifier.

```
void inserer(int *entier) {
    assert(NULL != entier);
    /* ... */
}
```

L'échec génère un fichier « core » analysable par GNU *gdb*.



Toujours laisser les appels à `assert()` dans le code.

Pour supprimer les lignes d'appel à `assert()` du code exécutable, il suffit de définir la constante `NDEBUG` en utilisant l'option « `-D` » lors de l'appel à *make*.

```
$ make -DNDEBUG
```

C'est une macro qui s'expande en `(void)0` avec `NDEBUG`, sinon la fonction `abort()` est appelée.

11.5.5 Résumé

La fonction `assert()`, des valeurs de retour ou des appels à `(f)printf()`, sont tous utilisés dans le cadre des tests automatiques, mais ils ne servent pas le même objectif.

- `(f)printf()` : trace du programme, vérification de valeurs pour le test.
- Valeurs de retour : utilisable lorsque l'erreur ne porte pas atteinte à l'intégrité du programme -- permettent de refuser simplement le service.

```
void inserer(int *entier) {
    if (NULL != entier) {
        /* ... */
    } else {
        /* ... */
    }
}
```

- `assert()` : violation d'une règle fondamentale qui empêche le logiciel de fonctionner.

```
void inserer(int *entier) {
    assert(NULL != entier);
    /* ... */
}
```

11.6 Tests

Un test est un ensemble de cas à tester et a pour but de mettre en évidence un ou des défauts. Les tests doivent être mis en oeuvre dès le commencement de la phase de développement. On parle même de « développement piloté par les tests » (*test driven development* pour TDD) lorsque ces derniers sont réalisés avant l'application.

11.6.1 Méthodes de tests

Tests de type *boîte transparente*

Les tests de type *boîte transparente* (ou tests « structurels ») sont des tests qui doivent être effectués dès qu'une fonction est écrite (ou mieux, avant en TDD). Pour cela, il faut créer un fichier « test-xxx.c » afin de tester les fonctions au fur et à mesure de leur écriture. On en profite pour placer des traces permettant d'afficher les valeurs de calculs intermédiaires. On insère aussi des appels à `assert()` de manière à contrôler les valeurs d'entrée de la fonction ainsi que les calculs.

Tests de type *boîte noire*

Les tests de type *boîte noire* (ou tests « fonctionnels ») sont en général effectués à la fin de la construction du logiciel. Ils doivent permettre de vérifier que le logiciel fonctionne :

- en condition normale;
- en condition limite;
- hors conditions.

11.6.2 Niveaux de tests

On rencontre entre autres les niveaux de tests suivants :

- **unitaires** : permettent au développeur de contrôler son travail au niveau d'une fonction;
- **fonctionnels** : vérifier tous les cas d'utilisation qu'ils soient positifs ou négatifs, les erreurs et les exceptions prévues, et valider le logiciel auprès du client;
- **non-régression** : vérifier que les nouveautés d'une itération n'ont pas d'impact sur les anciennes fonctionnalités -- les tests fonctionnels d'une version deviennent des tests de non-régression à l'itération suivante.

11.6.3 Tests et nombres aléatoires

Fonction qui donne un nombre aléatoire :

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

```
int rand();
```

Les nombres générés sont des nombres compris entre 0 et `RAND_MAX`.

```
int r;
```

```
r = a + (int) ((b - a) * (double)rand() / RAND_MAX);
```

Initialisation de la graine :

```
void srand(unsigned int)
```

Avec une graine identique, la série de nombres aléatoires est toujours la même, ce qui peut être très utile pour les tests. Pour avoir une graine pseudo-aléatoire :

```
/* #include <time.h> */
```

```
srand(time(NULL));
```

Il n'existe pas à proprement parler de logiciels permettant une génération automatique de tests. Les tests doivent donc être écrits par le développeur, en s'appuyant éventuellement sur des bibliothèques de tests telles que *minunit* (<https://github.com/siu/minunit>) ou *CUnit* (<http://cunit.sourceforge.net>). Par contre, il existe des logiciels permettant de gérer des suites de tests, puis de générer des rapports mettant en évidence les problèmes ainsi que leur origine, par exemple *CREST* (www.burn.im/crest/).

11.7 Optimisation du code

L'optimisation du code est obtenue avec les options du compilateur : `-O`, `-O2`, `-O3`, etc.



Il est nécessaire de compiler tous les modules avec cette option.

11.7.1 Outil : profileur

Il n'est pas toujours nécessaire d'optimiser toutes les parties d'un programme, mais seules les parties les plus utilisées doivent être optimisées. L'outil de profilage permet d'analyser la fréquence et le temps d'utilisation de fonctions lors de l'exécution du programme.

Description

Le profileur *GProf* fournit un bilan après une exécution particulière du programme. Dans ce bilan, on trouve :

- Le **profil à plat** : montre combien de temps le programme passe dans chacune des fonctions et combien de fois chacune d'elles est appelée. Cela permet de visualiser quelles sont les fonctions qui consomment le plus de ressources.

```
$ time cumulative-seconds self-seconds calls self-ms/call total ms/call name
```

- Le **graphe d'appel** : montre les graphes d'appel des fonctions et combien de fois une fonction en appelle une autre.

Mise en oeuvre

- **Étape 1** : Compiler les modules et le programme principal avec l'option `-pg` de *gcc*. Cette option ajoute du code dans l'exécutable produit.

```
$ gcc -pg -o prog prog.c
```

- **Étape 2** : Lancer le programme de façon habituelle. Une fois le programme terminé, un fichier « `gmon.out` » contenant le suivi de l'exécution a été généré.

```
$ prog 8
val=109601
$ ls
gmon.out prog prog.c
```

- **Étape 3** : Analyser le résultat en sortie avec la commande `gprof` :

```
$ gprof prog [gmon.out]
```

11.7.2 Analyse du temps d'exécution : la commande `time`

La commande `time` mesure les durées d'exécution d'une commande, ce qui est idéal pour connaître les temps de traitement. Elle retourne trois valeurs :

- `real` : durée totale d'exécution de la commande.
- `user` : durée du temps CPU nécessaire pour exécuter le programme.
- `system` : durée du temps CPU nécessaire pour exécuter les commandes liées au système d'exploitation (appels système au sein d'un programme).

```
$ time monprogramme
real      2m 2.65s
user      1m 1.48s
sys       0m 4.61s
```



On peut avoir une indication de la charge de la machine en effectuant le calcul `real / (user+system)`. Si le résultat est inférieur à 10, la machine dispose de bonnes performances, au-delà de 20 la charge de la machine est trop lourde.

11.7.3 Analyse du temps d'exécution : la fonction `clock()`

La fonction `clock()` donne le temps présent. Elle peut donc être utilisée pour chronométrer une partie du code :

```
#include <time.h>

clock_t begin, end;

begin=clock();
/* ... instructions ... */
end=clock();

printf("Running time: %f\n", (float)(begin-end)/CLOCKS_PER_SEC);
```

11.8 Notion de qualimétrie logicielle

La **qualité** d'un logiciel est associée à la mesure d'un certain nombre d'indicateurs qui vont nous renseigner sur la complétude des fonctionnalités du logiciel, sa fiabilité, sa tolérance aux pannes, sa portabilité, son ergonomie, etc.

Certains de ces indicateurs, comme l'ergonomie, peuvent être difficiles à définir. Nous nous limiterons donc ici aux indicateurs associés à la qualité du code source produit (« métriques »).



Différents standards de codage sont associés au langage C. On trouve en particulier : MISRA C, défini par la *Motor Industry Software Reliability Association* en 1998 (version actuelle : 2012), et le *SEI CERT C Coding Standard* (origine : DARPA -- défense américaine) [1]. Ces deux standards définissent des règles de « bonnes pratiques » de codage afin d'augmenter la sûreté des logiciels dans une démarche qualité.

11.8.1 Métriques

Plusieurs **métriques** ont été définies et permettent d'« estimer » la qualité d'un code source produit. On trouve des indicateurs simples tels que : le nombre total de lignes de codes, le nombre total de fonctions, le ratio lignes de codes/nombre de fonctions, le ratio lignes de commentaires/lignes de codes, etc., mais aussi des mesures plus complexes telles que la complexité cyclomatique, les métriques d'Halstead, l'indice de maintenabilité, etc.

Pour une description plus complète, on pourra se reporter à l'article en ligne de K. Lambertz [6].

11.8.2 Exemple de la complexité cyclomatique

La **complexité cyclomatique** d'une fonction a été définie par T. J. McCabe en 1976 [8]. Elle correspond au nombre de chemins linéairement indépendants qu'il est possible d'emprunter dans cette fonction. Ce nombre de chemins correspond à 1 (le chemin principal) + le nombre de points de branchement de la fonction (les structures de contrôle `if`, etc.).

La complexité cyclomatique d'une fonction est donc proportionnelle au nombre de points de branchement (au mieux, elle vaut 1). Plus sa valeur est élevée et plus la fonction sera difficile à comprendre et à maintenir.



Le nombre de tests unitaires pour une fonction donnée devrait être, dans l'idéal, égal à sa complexité cyclomatique : un test unitaire par chemin !

11.8.3 Outils

En fonction du langage utilisé, on trouvera un certain nombre d'outils aidant à la mesure de la qualité logicielle. Le tableau ci-dessous en regroupe quelques-uns.

Logiciel	Langage(s)	Open-source	Informations
Frama-C	C/C++	oui	--
ccccc	C	oui	--
Metrics	Java	oui	greffon Eclipse
gnatmetric	Ada	oui	inclus dans GNAT
Pynocle	Python	oui	--
Testwell CMT++	C/C++	non	--
Testwell CMT Java	Java	non	--
Metrics viewer	Csharp	non	--
LDRA	multilangages	non	--

Exercices

Exercice 11.1

Une matrice carrée en zigzag stocke les N^2 premiers entiers de telle sorte que les nombres augmentent séquentiellement lorsque l'on zigzague le long des anti-diagonales de la matrice comme le montre la figure suivante :

- Le code qui suit et que vous avez récupéré ne répond pas au problème.
- Utilisez les outils de mise au point (gdb) afin de le corriger.
- Réusinez le code afin de respecter les règles ODL et dans la mesure du possible, modularisez-le, par exemple, en créant les deux fonctions :

```
- void make_zigzag_matrix(int *matrix, int size);
- void display_zigzag_matrix(int *matrix, int size);
```

- Proposez des tests adaptés.

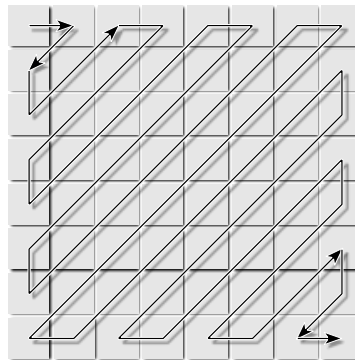


Fig. 11.2 : Parcours en zigzag de la matrice.

```

/* zigzag.c */
#include <stdio.h>
#include <stdlib.h>

int main(int c, char **v) {
    int i, j, m, n, *s;

    /* default size: 5 */
    if (c < 2 || ((m = atoi(v[1])) <= 0) m = 5;

    /* alloc array*/
    s = malloc(sizeof(int) * m * m);

    for (i = n = 0; i < m * 2; i++)
        for (j = (i < m) ? 0 : i-m+1; j <= i && j < m; j++)
            s[(i&1) ? j*m+i : (i-j)*m+j] = n++;

    for (i = 0; i < m * m; putchar((++i % m) ? ' ':'\n'))
        printf("%3d", s[i]);

    free(s);
    return 0;
}

```

Exercise 11.2

On souhaite tester une fonction de chargement d'images couleur depuis des fichiers de type « PGM ». Cette fonction prend en paramètre le nom du fichier contenant l'image et retourne une structure de données contenant les

pixels de l'image. Quel ensemble de données, parmi ceux proposés ci-dessous, est-il le plus adéquat pour valider la fonction?

- ☐ Des fichiers avec une belle image, une image horrible, une image libre de droits, une image non libre de droits, une image contenant des personnages, une image de paysage.
- ☐ Des fichiers avec une belle image, une image horrible, un fichier inexistant, un fichier PGM mal formé, un fichier avec un autre type d'image, un fichier non-image (par exemple un fichier Ms-Word).
- ☐ Des fichiers avec une très grande image, une très petite image, un fichier inexistant, un fichier PGM mal formé, un fichier avec un autre type d'image, un fichier non-image (par exemple un fichier Ms-Word).
- ☐ Des fichiers avec une très grande image, une très petite image, une image libre de droits, une image non libre de droits, une image contenant des personnages, une image de paysage.
- ☐ Aucune des réponses précédentes.

Exercice 11.3

À l'aide de la fonction `clock()`, comparer les temps d'exécution des trois fonctions ci-dessous pour des valeurs de n égales à 100, 1000, 10000, 100000 et 1000000.

```
/*
 * Returns the sum of the n integers. It takes O(1) time.
 */
int sum_O1(int n) {
    return n*(n-1)/2;
}

/*
 * Returns the sum of the n integers. It takes O(n) time (n * O(1)).
 */
int sum_On(int n) {
    int i, sum;

    sum = 0;

    for (i = 0; i < n; i++) {
        sum += i;
    }

    return sum;
}

/*
 * Returns the sum of the n integers. It takes O(n^2) time (n * n * O(1)).
 */
int sum_On2(int n) {
    int i, j, sum;

    sum = 0;

    for (i = 0; i < n; i++) {
        for (j = 0; j < i; j++) {
            sum++;
        }
    }
}
```

```
    }  
  
    return sum;  
}
```


12 Produit final

« If the automobile had followed the same development cycle as the computer, a Rolls-Royce would today cost \$100, get a million miles per gallon, and explode once a year, killing everyone inside. » Robert X. Cringely.

12.1 Entrées-sorties d'un programme

La ligne de commande complète de la fonction `main()` est la suivante :

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

Arguments de la ligne de commande

Par convention la fonction `main()` accepte jusqu'à trois arguments :

- le nombre d'arguments `argc` qui détient le nombre d'arguments y compris le nom du programme. Il y a toujours au moins un argument, c'est le nom du programme.
- l'adresse d'un tableau de pointeurs vers les arguments `argv[]`. L'intérêt de cet argument est :
 - de permettre au programme d'être résistant au changement de nom si par exemple l'application utilise le nom du programme pour afficher un message d'usage;
 - au contraire d'avoir deux comportements différents, le programme est nommé différemment. C'est le cas de `sh` et `bash` qui correspondent au même programme, mais qui ont un comportement différent selon la copie utilisée.
 - `argv[0]` contient l'adresse du premier caractère du nom du programme.
 - `argv[1]` contient l'adresse correspondant au premier argument, `argv[argc-1]` dernier argument.
- l'adresse d'un tableau de pointeurs vers les variables d'environnement `env[]`. Chaque élément `env[0]`, `env[1]`, etc. contient l'adresse d'une variable d'environnement (variables créées par la commande `set` ou `setenv` suivant les Unix). La dernière entrée contient `NULL`.

Vérification de la ligne de commande

Il faut toujours vérifier la ligne de commande d'un programme, c'est-à-dire le nombre d'arguments acceptés. En cas d'erreur il est nécessaire d'afficher un message présentant l'usage de la commande.

Dans le meilleur des cas, le logiciel accepte même des options permettant de paramétrer son exécution. Ainsi, Emacs accepte l'option `-nw` afin de le lancer en mode non graphique.

En général, les programmes devraient au moins accepter l'option `-h` permettant d'afficher l'usage de la commande.

Code d'erreur

Tout logiciel ou commande Unix retourne un code d'erreur. Par convention un programme qui se termine correctement doit retourner 0. Dans le cas contraire, il retourne une valeur x non nulle (numéro d'erreur purement conventionnel).

- Il est possible de récupérer le code d'erreur à l'aide de la variable Unix « `$?` ». Exemple : « `./mult; echo $?` » lance le programme `mult` puis affiche son code de retour.
- La valeur de retour d'un programme est donnée suite à un appel à `return x` ou `exit(x)`.
- La valeur de retour d'un programme est sur 8 bits et non signée [0, 255].

12.2 Organisation d'une distribution logicielle

Un logiciel correspond à une « distribution logicielle ». Les dossiers d'une distribution logicielle comporte les fichiers exécutables, les fichiers d'identification ainsi que la documentation (éventuellement les sources).

Organisation classique d'une distribution *open source*

- `src` : fichiers « `.c` » organisés en paquets;
- `lib` : toutes les bibliothèques « `.a` » ou « `.so` »;
- `include` : les fichiers d'entêtes;
- `bin` : les binaires exécutables;
- `etc` : tous les fichiers annexes;
- `doc` : les fichiers de la documentation.

Les fichiers d'identification :

- `COPIE.txt` (ou `COPYING.txt` ou `COPYRIGHT.txt`);
- `LISEZMOI.md` (ou `README.md`);
- `AUTEURS.txt` (ou `AUTHORS.txt`).

Les fichiers documentation :

- `ChangeLog` ou `NEWS.txt` (ne sont plus utiles avec les outils tels que *Git*);
- `INSTALLATION.txt` (ou `INSTALL`).

Les fichiers sources :

- Le script `configure` qui va produire le fichier `Makefile`, ou directement le fichier `Makefile`.
- « `.c` » et « `.h` ».

Conventions pour le `Makefile`

Pour une distribution donnée, on trouvera en général les options suivantes :

- `make` : utilise la cible `all` et doit lancer la compilation de tous les fichiers nécessaires à la construction du logiciel, y compris les bibliothèques, les documentations compilées, etc.;
- `make clean` : supprimer tous les fichiers intermédiaires et les exécutables créés à partir des sources et utilisés pendant la compilation -- conserve toutes les bibliothèques et les exécutables du logiciel;
- `make distclean` : supprimer tout ce qui a été créé et revient à l'état précédant la première compilation -- cela peut aller jusqu'à relancer le `./configure`;

- `make install` : copier tous les fichiers nécessaires au logiciel dans un autre dossier que celui d'installation ;
- `make uninstall` : supprimer tout ce qu'a généré la commande précédente.

Ainsi pour installer le logiciel on pourra utiliser :

- 1er cas : `make; make clean`
- 2e cas : `make; make install; make distclean`

Pour recompiler le logiciel, on utilisera :

- 1er cas : `make distclean; make; make clean`
- 2e cas : `make uninstall; make; make install; make clean`

12.3 Construction de l'archive de la distribution

La distribution logicielle sera la plupart du temps fournie sous forme d'une archive compressée en « .tgz » et/ou « .zip » (cf. chapitre 5).

12.4 Produit final

Fichier exécutable

La version finale de l'exécutable est compilée :

- sans les options `-g` et `-pg` ;
- strippée à l'aide de la commande Unix `strip` ou encore en appelant « `gcc -s` » (option de compilation, à ne pas confondre avec `-S`), ce qui permet de réduire la taille du code en supprimant toutes les instructions symboliques utilisables par le débogueur et donc devenues inutiles.

À ce niveau-là, il devient impossible d'utiliser GNU *gdb*.

Numérotation des versions

Gestion des versions avec les numéros de version (« `x.y.z` »).

- **x** : version ;
- **y** : révision ;
- **z** : corrections.

En particulier pour le numéro de version :

- 0 correspond à un brouillon ;
- 1 et supérieur correspondent aux versions diffusées.



Avec des outils de gestion des versions tels que Git que nous verrons dans le chapitre suivant, les numéros de versions seront indiqués à l'aide des « étiquettes ».

Gestion des versions

Le fichier historique des révisions est appelé « ChangeLog ». Toute révision d'un fichier source entraîne l'ajout d'une information dans celui-ci.

Il n'existe qu'un unique fichier « ChangeLog » par distribution. Celui-ci est présent dans le dossier correspondant à la distribution.

Exemple de révision dans « ChangeLog » :

```
2010-10-24  Jean Saigne <jsaigne@ensicaen.fr>  
src/operatorsP0/frequentiel/iffc.cpp : fix bug on line 192
```

```
2010-10-12  Jean Saigne <jsaigne@ensicaen.fr>  
src/operatorsP0/frequentiel/fft.cpp : fix bug on line 27
```

```
2005-10-24  Jean Saigne <jsaigne@ensicaen.fr>  
src/operatorsP0/frequentiel/base.cpp : fix bug on line 19
```



La gestion des versions d'un logiciel est réalisée actuellement par des outils puissants centralisés (*Subversion*) ou distribués (*Git*) comme nous allons le voir dans le chapitre suivant.

13 Gestion de version avec *Git*

« I'm an egotistical bastard, and I name all my projects after myself. First 'Linux', now 'Git' » Linus Torvalds.

13.1 Introduction

Git est un système distribué de gestion des versions développé par Linus Torvalds en 2005 (et oui le Linus de Linux). Il est utilisé pour suivre les modifications apportées à un ensemble de fichiers, généralement du code source, et permet de collaborer efficacement avec d'autres développeurs. Git est rapide, flexible, et convient à des projets de toutes tailles.

13.2 Configuration de Git

Git est en principe installé sur les machines de l'école. La première chose à faire est de configurer votre nom d'utilisateur et votre adresse de courriel :

```
$ git config --global user.name "Votre Nom"
$ git config --global user.email "votre.courriel@ecole.ensicaen.fr"
```

Vous pouvez ensuite vérifier votre configuration avec :

```
$ git config --list
```

13.3 Qu'est-ce qu'un dépôt Git ?

En général, un projet est associé à un dépôt Git. Un dépôt Git (*repository*) est une structure de données qui stocke toutes les versions des fichiers de ce projet. Il contient :

- Un dossier de travail : l'endroit où vous travaillez sur vos fichiers.
- Un index (appelé aussi zone de *staging*) : une zone intermédiaire où vous placez les modifications avant de les valider (*commit*).
- Un historique des validations : l'enregistrement de toutes les modifications apportées aux fichiers du projet.

Initialiser un dépôt Git

Pour créer un nouveau dépôt Git, utilisez la commande « `git init` » dans le dossier dans lequel vous souhaitez placer votre projet :

```
$ mkdir mon_projet
$ cd mon_projet
$ git init
```


Cette commande crée un sous-dossier « .git » dans le dossier de projet. Le dossier « .git » contient toutes les informations nécessaires pour le suivi des versions.

13.4 Commandes de base

Ajout de fichiers au dépôt

Pour ajouter des fichiers au suivi des versions, il faut utiliser la commande « `git add` » :

```
$ git add fichier1.txt      # Ajoute un fichier spécifique
$ git add .                 # Ajoute tous Les fichiers du dossier courant
```

Validation des modifications

La commande « `git commit` » va permettre de valider les modifications qui ont été réalisées dans le dépôt :

```
$ git commit -m "Message de commit"
```

Le message de validation doit être court, descriptif et expliquer les changements apportés. On tentera dans la mesure du possible de réaliser des validations très régulièrement afin de conserver un historique significatif.

Par exemple :

- Toutes les 15 à 30 minutes pour des changements simples et isolés comme la correction d'une petite erreur.
- Toutes les 1 à 2 heures pour des modifications de taille modérée (par exemple, l'ajout d'une nouvelle fonctionnalité.
- Toutes les 4 à 6 heures pour des modifications plus importantes et complexes comme le réusinage du code.

Visualisation de l'état du dépôt

Pour vérifier l'état du dépôt à tout instant, il faut utiliser la commande « `git status` » :

```
$ git status
```

Cette commande affiche les fichiers modifiés, ceux ajoutés au suivi et ceux en attente de validation.

Historique des validations

La commande « `git log` » permet de visualiser l'historique des validations qui ont été réalisées pour le projet :

```
$ git log
```

Pour un historique plus compact, on utilisera la commande « `git log --oneline` ».

Comparaison des modifications

Afin de visualiser les différences entre les fichiers modifiés et la dernière version validée, on utilise la commande « `git diff` » :

```
$ git diff
```

13.5 Branches dans Git

Les branches permettent de travailler sur différentes versions de votre projet simultanément. La branche par défaut s'appelle `main` (ou `master`).

Création d'une nouvelle branche

Pour créer une nouvelle branche, on utilise la commande « `git branch` » :

```
$ git branch nouvelle_branche
```

Changement de branche

La commande « `git checkout` » permet de basculer sur une autre branche :

```
$ git checkout nouvelle_branche
```

Fusion de branches

Pour fusionner une branche dans une autre, on bascule sur la branche de destination et on lance la commande « `git merge` » :

```
$ git checkout main
$ git merge nouvelle_branche
```

13.6 Travail avec un dépôt distant

Clonage d'un dépôt distant

La commande « `git clone` » permet de cloner un dépôt distant :

```
$ git clone https://gitlab.ecole.ensicaen.fr/utilisateur/mon_projet.git
```

Ajout d'un dépôt distant

Pour ajouter un dépôt distant, on utilise la commande « `git remote add` » :

```
$ git remote add origin https://gitlab.ecole.ensicaen.fr/utilisateur/mon_projet.git
```

Publication des modifications

La commande « `git push` » permet de publier les validations vers un dépôt distant :

```
$ git push origin main
```

Récupération de modifications distantes

La commande « `git pull` » permet de récupérer les modifications faites sur un dépôt distant :

```
$ git pull origin main
```

Création d'étiquettes

Les étiquettes permettent de facilement retrouver des versions spécifiques. La création d'une étiquette est réalisée avec la commande « `git tag` » :

```
$ git tag -a v1.0 -m "Version 1.0" # nouvelle étiquette avec son nom et sa description
$ git push origin v1.0
```

13.7 Gestion des conflits

Les conflits surviennent lorsque deux modifications incompatibles sont apportées au même fichier. Pour résoudre un conflit, il est nécessaire de :

1. Identifiez le conflit avec « `git status` ».
2. Éditez le fichier concerné pour résoudre le conflit manuellement.
3. Ajoutez le fichier corrigé avec « `git add` ».
4. Validez la résolution du conflit avec « `git commit` ».

13.8 Exemple complet

Créer un dépôt et y pousser du code

1. Créer un nouveau projet sur le GitLab de l'ENSICAEN :
 - Allez sur [GitLab](#).
 - Connectez-vous avec vos identifiants.
 - Cliquez sur le bouton + à gauche et en haut (« Créer un nouveau projet »).
 - Sélectionnez « Créer un projet vide ».
 - Donnez un nom à votre projet (par exemple, « MonPremierDepot »), décochez la case « Initialiser le dépôt avec un README », puis cliquez sur le bouton « Créer le projet ».
 - La page sur laquelle vous vous trouvez est celle du dépôt. Le dépôt est vide, mais Gitlab vous informe des commandes que vous pouvez utiliser. Copiez, puis collez dans un terminal celles qui permettent de configurer votre nom et votre adresse de courriel.

2. Initialiser un dépôt Git local :

```
$ mkdir MonPremierDepot
$ cd MonPremierDepot
$ git init
```

3. Ajouter un fichier README.md localement :

```
$ echo "# Mon Premier Dépôt" > README.md
$ git add README.md
$ git commit -m "Ajout du fichier README.md"
```

- Le fichier README.md est donc ajouté et validé pour ce dépôt.

4. Pousser le dépôt local vers GitLab :

- Sur GitLab, cliquez sur le bouton « Code » et copiez l'URL de la zone « Cloner avec HTTPS ».

- L'URL ressemblera à quelque chose comme :

```
https://gitlab.ecole.ensicaen.fr/utilisateur/monpremierdepot.git
```

```
$ git remote add origin https://gitlab.ecole.ensicaen.fr/utilisateur/monpremierdepot.git
$ git branch -M master
$ git push -u origin master
```

puis renseignez votre nom d'utilisateur et votre mot de passe.

- Sur Gitlab, rechargez la page et vérifiez la présence du fichier README.md.

13.8.1 Créer une branche, y effectuer des modifications et les fusionner

1. Créer une nouvelle branche

```
$ git checkout -b nouvelle_fonctionnalite
```

2. Modifier le fichier README.md :

```
$ echo "## Nouvelle Fonctionnalité" >> README.md
$ git add README.md
$ git commit -m "Ajout de la section 'Nouvelle Fonctionnalité' dans README.md"
```

3. Pousser la branche vers GitLab :

```
$ git push -u origin nouvelle_fonctionnalite
```

Vous pouvez vérifier en rechargeant la page du dépôt Gitlab, qu'une nouvelle branche existe en cliquant sur la liste déroulante « master ».

4. Créer une requête de fusion (*merge request*) sur GitLab :

- Depuis le dépôt sur GitLab, cliquez à gauche sur « Requêtes de fusion », puis cliquez sur le bouton « Nouvelle requête de fusion » qui est apparu.
- Dans la zone « Branche source », sélectionnez « nouvelle_fonctionnalite » comme branche et laissez « master » comme branche cible.
- Cliquez sur « Comparer les branches et continuer ».
- Dans la page qui apparaît, ajoutez un titre et une description, puis cliquez sur « Créer requête de fusion ».

5. Fusionner la branche sur GitLab :

- Une fois la requête soumise, vous pouvez la fusionner en cliquant sur « Fusionner » (en modifiant éventuellement les différents champs si nécessaire).

13.9 Conclusion

Git est un donc outil puissant et flexible pour la gestion de versions. Avec les commandes de base présentées dans ce chapitre, vous êtes maintenant prêt à démarrer et gérer vos projets avec Git. Pour aller plus loin, consultez la documentation officielle sur git-scm.com.

Exercices

Exercice 13.1

1. Initialiser un dépôt Git local :
 - Créez un nouveau dossier nommé `ProjetExercice1`.
 - Initialisez un dépôt Git dans ce dossier.
 - Créez un fichier `index.html` avec un contenu simple (par exemple, une structure HTML de base).
 - Ajoutez ce fichier au suivi de version.
 - Validez le fichier avec un message approprié.
2. Pousser les modifications vers un dépôt distant :
 - Créez un nouveau projet sur GitLab nommé `ProjetExercice1`.
 - Ajoutez ce dépôt distant à votre dépôt local.
 - Poussez vos modifications vers le dépôt distant.

Exercice 13.2

1. Créer et changer de branche :
 - Dans le dossier `ProjetExercice1`, créez une nouvelle branche nommée `nouvelle_fonctionnalite`.
 - Basculez sur cette branche.
2. Modifier et valider les fichiers :
 - Modifiez le fichier `index.html` en ajoutant un paragraphe avec le texte « Nouvelle fonctionnalité ajoutée ».
 - Ajoutez et validez cette modification avec un message approprié.
3. Fusionner la branche :
 - Basculez de nouveau sur la branche `master`.
 - Fusionnez la branche `nouvelle_fonctionnalite` avec la branche `master`.
 - Poussez les modifications vers le dépôt distant.

Exercice 13.3

1. Préparer l'environnement pour un conflit :
 - Clonez le dépôt `ProjetExercice1` dans un nouveau dossier nommé `ProjetExercice1_conflit`.
 - Dans le dépôt original (`ProjetExercice1`), créez une nouvelle branche nommée `conflit`.
 - Modifiez le fichier `index.html` en changeant le contenu du paragraphe existant et validez la modification.
 - Poussez la branche `conflit` vers le dépôt distant.
2. Créer un conflit :
 - Dans le dossier `ProjetExercice1_conflit`, modifiez le fichier `index.html` en changeant également le contenu du paragraphe existant mais avec un texte différent et validez la modification.
 - Tentez de fusionner la branche `conflit` depuis le dépôt distant dans votre branche locale `master` pour provoquer un conflit.
3. Résoudre le conflit :

- Résolvez manuellement le conflit en éditant le fichier `index.html`.
- Ajoutez et validez le fichier résolu.
- Poussez la résolution vers le dépôt distant.


14 Éléments de réponse aux exercices

Exercices du chapitre 5

Exercice 5.1

À simplifier avec l'instruction *for* du *shell* :

```
$ mkdir -p d1/sd2/ssd3/sssd4/
$ cp /home/public/tp_od1/tp01/loremipsum.txt d1/loremipsum.txt
$ cp /home/public/tp_od1/tp01/loremipsum.txt d1/sd2/loremipsum.txt
$ cp /home/public/tp_od1/tp01/loremipsum.txt d1/sd2/ssd3/loremipsum.txt
$ cp /home/public/tp_od1/tp01/loremipsum.txt d1/sd2/ssd3/sssd4/loremipsum.txt
$ chmod 0644 d1/loremipsum.txt
...
$ rm -r d1/* ; rmdir d1
```

Remarque : pensez à utiliser la touche  pour la complétion.

Exercice 5.2

```
$ rm ./-r!
```

Exercice 5.3

```
$ vi io.c & top &
[1] 22866
[2] 22867
$ jobs
[1]+  Arrêté                vi io.c
[2]-  Arrêté                top
$ fg %1 %1                # cela lance vi depuis lequel on tape C-z
vi io.c

[1]+  Arrêté                vi io.c
$ bg %1
[1]+ vi io.c &

[1]+  Arrêté                vi io.c
$ kill %1

[1]+  Arrêté                vi io.c
$ fg %2                    # fait passer "top" à l'avant-plan
```


Exercice 5.4

Ctrl+ b \$ + projetQuizz
 Ctrl+ b %
 Ctrl+ b "
 Ctrl+ b "
 Ctrl+ b ←
 Ctrl+ b "

Exercice 5.5

- umask 022 → rwxr-xr-x (supprime le droit w pour le groupe et les autres).
- umask 077 → rwx----- paranoïa
- umask 026 → rwxr-x--x

Exercices du chapitre 6

Exercice 6.2

C'est ainsi qu'un soir d'hiver, Arsène Lupin me raconta l'histoire de son arrestation. Le hasard d'incidents dont j'écrirai quelque jour le récit avait noué entre nous des liens... dirai-je d'amitié ? Oui, j'ose croire qu'Arsène Lupin m'honore de quelque amitié, et que c'est par amitié qu'il arrive parfois chez moi à l'improviste, apportant, dans le silence de mon cabinet de travail, sa gaieté juvénile, le rayonnement de sa vie ardente, sa belle humeur d'homme pour qui la destinée n'a que faveurs et sourires.

Exercice 6.3

Il me dévisagea profondément, puis il me dit, les yeux dans les yeux :

– Arsène Lupin, n'est-ce pas ?

Je me mis à rire.

– Non, Bernard d'Andrézy, tout simplement.

– Bernard d'Andrézy est mort il y a trois ans en Macédoine.

Exercice 6.4

Lagardère secoua la tête.

– J'aimerais mieux Carrigue et mes gens avec leurs carabines, répliqua-t-il.

Il s'interrompit tout à coup pour demander :

– Êtes-vous venu seul ?

– Avec un enfant : Berrichon, mon page.

– Je le connais ; il est leste et adroit. S'il était possible de le faire venir...

Nevers mit ses doigts entre ses lèvres, et donna un coup de sifflet retentissant.

Un coup de sifflet pareil lui répondit derrière le cabaret de la Pomme-d'Adam.

– La question est de savoir, murmura Lagardère, s'il pourra parvenir jusqu'à nous.

– Il passerait par un trou d'aiguille ! dit Nevers.

Exercice 6.5

1. `Ctrl`+`c` (copie du texte dans le presse-papier), puis `vim fic.txt`. On passe en mode insertion (`i`) et on colle le presse-papier `Ctrl`+`v`. Enfin on enregistre (`:w`).
2. Pour enregistrer sous `:w fic1_2.txt` et pour changer quelques caractères en majuscules on utilise la commande `gU` et on se déplace avec les flèches ou bien `h`, `j`, `k` et `l`.
3. Ouvrir à nouveau « `fic1.txt` » : « `:e fic1.txt` » et afficher les deux tampons dans deux fenêtres verticales : `:vert`.
4. Afin de visualiser les différences entre les fichiers, on passe en mode « `diff` » en entrant `:diffthis` dans chaque *viewport* (on passe de l'un à l'autre en entrant `Ctrl`+`w` `w`).
5. On se place en début de ligne au début du fichier, on entre `gU` et on appuie sur la flèche `↓`, puis on réitère jusqu'en fin de fichier. La commande `u` permet de revenir en arrière.

Exercices du chapitre 7

Exercice 7.1

Nous pouvons utiliser la redirection « `>` » si le fichier texte a vocation à être effacé, ou « `>>` » sinon :

```
#!/bin/sh
read phrase
echo $phrase > texte.txt
```

Exercice 7.2

Nous utilisons la syntaxe `$((opérande opération opérande))` :

```
#!/bin/sh

echo -n "Entrer le premier nombre : "; read x
echo -n "Entrer le second nombre : "; read y
somme=$((x + y))
difference=$((x - y))
produit=$((x * y))
rapport=$((x / y))
reste=$((x % y))

# Affichage des résultats
echo "Somme : $somme"
echo "Différence : $difference"
```

```
echo "Produit : $produit"
echo "Rapport : $rapport"
echo "Reste : $reste"
```

Exercice 7.3

La réalisation du fichier archive met en oeuvre la commande `tar` :

```
#!/bin/sh
echo "Entrez le nom du dossier à archiver"
read dossier
sauvegarde=./backup-$(date +%d-%m-%y).tar.gz
tar -czf $sauvegarde $dossier
```

Exercice 7.4

Le script « `pass_generator.sh` » pourra s'écrire ainsi :

```
#!/bin/bash
echo "Here are " $1 " passwords with a length of " $2
cat /dev/urandom | tr -dc '!@#%^&*()_A-Z-a-z-0-9' | fold -w$2 | head -$1
```

Exercice 7.5

```
$ echo "a(1)*4" | bc -l
```

Exercices du chapitre 8

Exercice 8.1

- L2 : manque un espace entre `)` et `{`.
- L3 : types différents (tableau et `double`).
- L3 : nom de variable non explicite.
- L6 : manque espace entre `for` et `(`.
- L6 : aération souhaitable (mais non obligatoire) autour des opérateurs.
- L7 : `{` devrait se trouver en L6.
- L9 : manque une indentation.
- L9 : le passage à la ligne doit se faire avant l'opérateur `*`.
- L11 : `}` doit être indentée à droite.
- L14 : en ANSI un passage à la ligne est attendu en fin de fichier.

Exercice 8.2

- L2 : espace en trop entre `main` et `(`.
- L3 : types différents (tableau et `int`).
- L3 : nom de variable non explicite.
- L5 : manque espace entre `for` et `(`.
- L5 : aération inégale `--` souhaitable (mais non obligatoire) autour des opérateurs.
- L6 : `{` devrait se trouver en L5.

- L7 : manque un espace entre) et {.
- L8 : le passage à la ligne doit se faire avant l'opérateur *.
- L14 : en ANSI un passage à la ligne est attendu en fin de fichier.

Exercice 8.3

1. Si vous avez accès à *Artistic Style*, utilisez la commande `gggqG` (cf. section « Ajout de fonctionnalités à Vim ») afin de remettre en forme le code de « `factorial.c` ». Pour insérer en début de fichier : « `:r header1.c` ». Attention, n'oubliez aucun cartouche ni aucune balise, sous peine de ne pas obtenir une documentation correcte.
2. Afin d'ouvrir un nouvel onglet avec le fichier `Doxyfile`, on entre : « `:tabnew Doxyfile` » que l'on éditera comme ci-dessous. Pour exécuter les commandes de production de la documentation, on utilise : « `:!doxygen Doxyfile` ».

```
PROJECT_NAME = "The factorial program"
OUTPUT_DIRECTORY = .
INPUT = factorial.c
GENERATE_HTML = YES
GENERATE_LATEX = NO
GENERATE_RTF = NO
```

Exercice 8.4

Si vous avez accès à *Artistic Style*, utilisez la commande `gggqG` (cf. section « Ajout de fonctionnalités à Vim »).

Exercices du chapitre 9

Exercice 9.1

- `gcc -Wall ex01_warnings.c` : `ages` n'est pas initialisé (L33).
- `gcc -Wall -Wextra ex01_warnings.c` : *idem* + comparaison entre entier signé et non signé (L32).
- `gcc -Wall -Wextra -ansi ex01_ansi_pedantic.c` : les commentaires // (L30) ne sont pas ANSI/ISO C90 → erreur de compilation + 2 avertissements (`ages` non utilisée, `voting_age` initialisée, mais non utilisée).
- `gcc -Wall -Wextra -ansi -pedantic ex01_ansi_pedantic.c` : *idem* + un avertissement sur l'utilisation d'une variable (`argc`) pour dimensionner le tableau `ages`.

Exercice 9.5

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

int square(int number);

int main(void) {
    int index;

    for (index=0; index < 10; index++) {
        printf("%d * %d = %d\n", index, index, square(index));
```

```

    }

    return EXIT_SUCCESS;
}

int square(int number) {
    return number * number;
}

```

Exercice 9.6

1. vim -O bonjour.c bonsoir.c
2. gcc -Wall -Wextra -ansi -pedantic bonjour.c bonsoir.c -c
3. ar crv libjoursoir.a bonjour.o bonsoir.o

Exercice 9.7

1. vim -O bonjour.c bonsoir.c
2. gcc -Wall -Wextra -ansi -pedantic -fPIC bonjour.c bonsoir.c -c
3. gcc -shared -o libjoursoir.so bonjour.o bonsoir.o

Ne pas oublier de mettre à jour la variable d'environnement LD_LIBRARY_PATH.

Exercices du chapitre 10

Exercice 10.1

```

CC=gcc
CPPFLAGS=-I.
CFLAGS=-Wall -Wextra

```

```
all: worms
```

```
exo1: exo1.o erreur.o
```

```
    $(CC) $^ -o $@
```

```
exo1.o: exo1.c erreur.h
```

```
    $(CC) $(CPPFLAGS) $(CFLAGS) -c $<
```

```
erreur.o: erreur.c erreur.h
```

```
    $(CC) $(CPPFLAGS) $(CFLAGS) -c $<
```

```
clean:
```

```
    rm *.o
```

```
distclean: clean
```

```
    rm exo1
```

```
.PHONY: all clean
```

Exercice 10.2

```
CC=gcc
CPPFLAGS=-I.
CFLAGS=-Wall -Wextra

all: qmotor

qmotor: qmotor.o util.o common.o libmotor.a
    $(CC) $^ -o $@

qmotor.o: qmotor.c motor.h common.h util.h
    $(CC) $(CPPFLAGS) $(CFLAGS) -c $<

util.o: util.c util.h common.h
    $(CC) $(CPPFLAGS) $(CFLAGS) -c $<

common.o: common.c common.h
    $(CC) $(CPPFLAGS) $(CFLAGS) -c $<

clean:
    rm *.o

distclean: clean
    rm qmotor

.PHONY: all clean
```

Exercice 10.3

```
CC=gcc
CPPFLAGS=-I.
CFLAGS=-Wall -Wextra
LDFLAGS=-lncurses

all: worms

worms: main.o worms.o helper.o
    $(CC) $^ -o $@ $(LDFLAGS)

main.o: main.c helper.h worms.h
    $(CC) $(CPPFLAGS) $(CFLAGS) -c $<

worms.o: worms.c worms.h helper.h
    $(CC) $(CPPFLAGS) $(CFLAGS) -c $<

helper.o: helper.c helper.h
    $(CC) $(CPPFLAGS) $(CFLAGS) -c $<

clean:
    rm *.o
```

```
distclean: clean
    rm worms
```

```
.PHONY: all clean
```

Exercices du chapitre 11

Exercice 11.1

```
void make_zigzag_matrix(int *matrix, int size) {
    int i;
    int j;
    int n;
    int index;

    for (i = n = 0; i < size * 2; i++) {
        if (i < size) {
            j = 0;
        } else {
            j = i - size + 1;
        }
        for (j; j <= i && j < size; j++) {
            index = (i & 1) ? j*(size-1)+i : (i-j)*size+j;
            matrix[index] = n++;
        }
    }
}
```

Exercice 11.2

- ☐ Des fichiers avec une belle image, une image horrible, une image libre de droits, une image non libre de droits, une image contenant des personnages, une image de paysage.
- ☐ Des fichiers avec une belle image, une image horrible, un fichier inexistant, un fichier PGM mal formé, un fichier avec un autre type d'image, un fichier non-image (par exemple un fichier Ms-Word).
- ☒ Des fichiers avec une très grande image, une très petite image, un fichier inexistant, un fichier PGM mal formé, un fichier avec un autre type d'image, un fichier non-image (par exemple un fichier Ms-Word).
- ☐ Des fichiers avec une très grande image, une très petite image, une image libre de droits, une image non libre de droits, une image contenant des personnages, une image de paysage.
- ☐ Aucune des réponses précédentes.

Exercice 11.3

```
#include <time.h>

int main(void) {
    clock_t begin;
    clock_t end;
    int sum;

    for (i = 100; i <= 1000000; i *= 10) {
```

```
begin=clock();
sum = sum_01(i);
end=clock();
printf("sum_01(%d): %f\n", i, (float)(fin-deb)/CLOCKS_PER_SEC);

/* ... */
}

return 0;
}
```

Exercices du chapitre 13

Exercice 13.1

Initialiser le dépôt :

```
$ mkdir ProjetExercice1
$ cd ProjetExercice1
$ git init
$ echo "<!DOCTYPE html><html><head><title>Exercice
↳ 1</title></head><body><p>Bienvenue!</p></body></html>" > index.html
$ git add index.html
$ git commit -m "Ajout du fichier index.html"
```

Pousser le dépôt :

```
$ git remote add origin https://gitlab.ecole.ensicaen.fr/utilisateur/ProjetExercice1.git
$ git branch -M master
$ git push -u origin master
```

Exercice 13.2

Créer et changer de branche :

```
$ git checkout -b nouvelle_fonctionnalite
```

Modifier et valider les fichiers :

```
$ echo "<p>Nouvelle fonctionnalité ajoutée</p>" >> index.html
$ git add index.html
$ git commit -m "Ajout de la section 'Nouvelle fonctionnalité' dans index.html"
```

Fusionner la branche :

```
$ git checkout master
$ git merge nouvelle_fonctionnalite
$ git push origin master
```

Exercice 13.3

Préparer l'environnement pour un conflit :


```
$ git clone https://gitlab.ecole.ensicaen.fr/utilisateur/ProjetExercice1.git ProjetExercice1_conflit
$ cd ProjetExercice1
$ git checkout -b conflit
$ echo "<p>Modification sur la branche conflit</p>" > index.html
$ git add index.html
$ git commit -m "Modification sur la branche conflit"
$ git push -u origin conflit
```

Créer un conflit :

```
$ cd ../ProjetExercice1_conflit
$ echo "<p>Modification locale pour créer un conflit</p>" > index.html
$ git add index.html
$ git commit -m "Modification locale pour créer un conflit"
$ git pull origin conflit
```

Résoudre le conflit :

```
$ # Editer Le fichier index.html pour résoudre Le conflit manuellement
$ git add index.html
$ git commit -m "Résolution du conflit dans index.html"
$ git push origin master
```

Index

Anomalie, 117

ANSI, 86

astyle, 80

Bibliothèque, 96

 Dynamique, 96–98

 Statique, 96, 97

Bogue, *voir* Anomalie

Cartouches d'un programme, 73

Chaîne de compilation, 85

 Assemblage, 89

 Commandes

 as, 89

 cpp, 91

 gcc, 85

 ld, 90, 95

 ranlib, 98

 Edition des liens, 90, 95

 Optimisation, 87

 Précompilation, 91

 Traduction, 86

Commande shell

 ulimit, 118

Commande Unix, 23

 apropos, 25

 ar, 97

 basename, 27

 bc, 69

 bunzip2, 32

 bzip2, 32

 cat, 29

 chgrp, 27

 chmod, 27

 chown, 27

 clear, 24

 compress, 32

 cp, 27

 curl, 33

 cut, 29

 date, 35

 dd, 26

 diff, 29, 30

 dirname, 27

 dos2unix, 29

 du, 25

 echo, 70

 file, 26

 find, 27

 finger, 34

 fold, 70

 grep, 29

 groups, 34

 gunzip2, 32

 gzip, 32

 head, 29

 hexdump, 91

 iconv, 26

 info, 25

 kill, 32

 killall, 32

 less, 26

 ln, 27

 ls, 25

 lsof, 35

 make, 101, 103

 man, 25

 mkdir, 25

 more, 26

 mv, 27

 od, 91

 ps, 32

 rm, 27

 rmdir, 25

 rwho, 34

 scp, 33

 sftp, 33

 sleep, 32

 sort, 29

 ssh, 33

 strip, 135

- stty, 17
- tail, 29
- talk, 34
- tar, 32
- tee, 29, 60
- time, 32, 127
- top, 32
- touch, 26, 103
- tr, 29, 70
- uname, 35
- uncompress, 32
- unix2dos, 29
- unzip, 32
- users, 34
- wc, 29
- wget, 33
- whereis, 31
- which, 31
- who, 34
- whoami, 34
- xkill, 32
- zip, 32
- Commentaire, 72
- Descripteur de fichier, 24
- Directive
 - define, 93
 - elif, 93
 - else, 93
 - endif, 93
 - error, 94
 - if, 93
 - include, 91
 - warning, 94
- Dossier, 19
- Doxygen, 72, 81
- Droits d'accès, 22
- Débogueur, *voir* Dévermineur
- Dévermineur, 118, 122
- Editeur, *voir* Vim
- Entrée standard (stdin), 24
- Erreur d'algorithme, 122
 - assert, 123
 - Trace, 122
- Erreur d'exécution, 118
- Erreur de bus, *voir* Erreur d'exécution
- Erreur de segmentation, *voir* Erreur d'exécution
- Expressions régulières, 30
- Fichier core, *voir* Erreur d'exécution
- Fichiers de configuration, 56
 - .bash_profile, 56
 - .bashrc, 57
 - .profile, 56, 97
 - /etc/profile, 56
- Fuite de mémoire, 120
- Gcc, 85
- gdb, *voir* Dévermineur
- Gestion des versions, 136, 137
- Git, *voir* Gestion des versions
 - Ajout d'un dépôt distant, 139
 - Ajout de fichiers, 138
 - Changement de branche, 139
 - Clonage d'un dépôt distant, 139
 - commit, *voir* Validation de modifications
 - Comparaison des modifications, 138
 - Création de branches, 139
 - Etat du dépôt, 138
 - Etiquette, 140
 - Fusion de branches, 139
 - Gestion de conflit, 140
 - Historique des validations, 138
 - Publication des modifications, 139
 - Récupération de modifications, 139
 - Validation de modifications, 138
- gtop, *voir* top
- Génération de la documentation, *voir* Doxygen
- htop, *voir* top
- httptrack, 34
- Interpréteur de commandes, 55
- make
 - Options, 110
- Makefile, 102
 - Cible, 103
 - all, 105
 - clean, 105
 - distclean, 105
 - Type fichier, 105
 - Type identificateur, 105
- Commande, 104
- Directive, 109
 - .DEFAULT, 109
 - .IGNORE, 109
 - .PHONY, 110
 - .PRECIOUS, 110

- .SILENT, 110
- .SUFFIXE, 108
- include, 109
- Dépendance, 104
- Macro interne, 106
- Macro interne classique, 106
- Macro interne prédéfinie, 107
- Règle, 102
- Règle explicite, 105
- Règle implicite, 107, 108
- wildcard, 110
- Multiplexeur de terminal, 36
- Optimisation du code, 126
- Profileur, *voir* Optimisation du code
- Protection des fichiers, 21
- Qualimétrie, 127
- Qualité du code, *voir* Qualimétrie
- Répertoire, *voir* Dossier
- Shell, *voir* Interpréteur de commandes
 - Commandes
 - alias, 57
 - bg, 32, 68
 - cd, 25, 68
 - echo, 56
 - eval, 68
 - exec, 60, 68
 - exit, 56, 68
 - export, 56, 61, 68
 - fg, 32, 68
 - history, 24, 35, 58
 - jobs, 32, 68
 - pwd, 25, 68
 - read, 68
 - set, 57, 68
 - shift, 62
 - test, 62
 - trap, 68
 - type, 31, 68
 - ulimit, 68
 - umask, 35, 68
 - unalias, 57
 - unset, 61, 68
 - Fonctions, 67
 - Métacaractères, 68
 - Opérateurs logiques, 64
 - Script, 62
 - Structures de contrôle, 64
 - Substitution, 61
 - Variable, 60
 - Variable système, 61
- Sortie d'erreur (stderr), 24
- Sortie standard (stdout), 24
- Tests, 124
- tmux, *voir* Multiplexeur de terminal
- valgrind, *voir* Fuite de mémoire
- Variable d'environnement, 61
 - HISTSIZE, 57
 - HOME, 20, 68
 - LD_LIBRARY_PATH, 56, 61, 97
 - MANPATH, 25, 56, 61
 - PATH, 21, 31, 56, 61
 - PS1, 56
 - PS2, 56
 - SHELL, 56, 104
- Vim, 41
 - Annuler, 47
 - astyle, *voir* astyle
 - Chiffrer, 50
 - Coller, 46
 - Comparer des fichiers, 49
 - Compilation, 98
 - Configuration, 50
 - Convertir en majuscule, 47
 - Convertir en minuscule, 47
 - Copier, 46
 - Couper, 46
 - Déplacement curseur, 44
 - Enregistrer, 45
 - Fusionner des fichiers, 49
 - Indenter, 50
 - Lancement de commandes externes, 49
 - Mode insertion, 44
 - Modes, 42
 - Multifenêtrage, 48
 - Onglets, 48
 - Quitter, 45
 - Rechercher, 49
 - Remplacer, 46, 49
 - Tampons, 47

Bibliographie

- [1] [SEI CERT c coding standard : Rules for developing safe, reliable, and secure systems](#), Pennsylvanie, USA, 2016.
- [2] C. Albing, J.P. Vossen, C. Newham, Bash cookbook, O'Reilly, 2007.
- [3] J.-P. Armspach, P. Colin, F. Ostré-Waerzeggers, UNIX : Initiation et utilisation, 3rd ed., Dunod, 2005.
- [4] D. Cameron, J. Elliatt, M. Loy, E.S. Raymond, B. Rosenblatt, Learning GNU emacs, 3rd ed., O'Reilly, 2005.
- [5] B.W. Kernighan, D.M. Ritchie, Le langage c (norme ANSI), 2nd ed., Dunod, 2014.
- [6] K. Lambertz, [Complexité et qualité](#), (2007) 1–8.
- [7] N. Matloff, P.J. Salzman, The art of debugging with GDB, DDD and eclipse, 1st ed., No Starch Press, 2008.
- [8] T.J. McCabe, A complexity measure, IEEE Transactions on Software Engineering SE-2 (1976) 308–320.
- [9] R. Mecklenburg, Managing projects with GNU make, 3rd ed., O'Reilly, 2005.
- [10] A. Robbins, L. Lamb, E. Hannah, Learning the vi and vim editors, 7th ed., O'Reilly, 2008.
- [11] W.E. Shotts, The linux command line - a complete introduction, No Starch Press, 2012.
- [12] R. Stallman, [Using the GNU compiler collection](#), GNU Press, Boston, MA, USA, 2003.
- [13] A.S. Tanenbaum, H. Bos, Modern operating systems, 5th ed., Pearson, 2023.
- [14] B. Ward, How linux works - what every superuser should know, 2nd ed., No Starch Press, 2015.

