Une introduction au Langage C

Jean-Jacques Girardot, Marc Roelens

Septembre 2004

Première partie

Cours

Introduction

Avant-Propos

L'objectif général de ce cours est l'apprentissage de méthodes et outils permettant de résoudre des problèmes de l'ingénieur nécessitant un traitement automatisé de l'information. L'une des finalités est également de proposer aux élèves entrant en première année à l'École des Mines de Saint-Etienne une mise à niveau en programmation.

Il s'agit donc, dans un premier temps, d'analyser le problème, d'en proposer un modèle, et d'établir une méthode de résolution possible. Cette phase relève des techniques générales de modélisation. Dans un second temps, le modèle et la méthode de résolution choisis sont implémentés en utilisant efficacement la technologie informatique retenue : l'écriture d'un programme de traitement en langage *C*.

0.1 Préambule

0.1.1 Le support de cours

Ce document est une brève introduction à la programmation. Pour des raisons qui sont détaillées plus loin, cette introduction s'effectue au travers du langage *C*. Le langage de programmation particulier utilisé comme support de la pensée dans un cours d'informatique ne devrait jouer qu'un rôle assez secondaire vis à vis des concepts enseignés. Ce n'est malheureusement presque jamais le cas, et, en particulier dans le cas des « initiations » à l'informatique, ce choix du langage joue un rôle prépondérant. L'expérience montre que l'apprenant est en effet rapidement préoccupé (voire envahi) par les seuls aspects de mise en œuvre du langage choisi, et que les concepts, parfois de haut vol, que l'enseignant vise à mettre à sa disposition sont largement ignorés (lorsqu'ils ne sont pas traités par le mépris).

Nous voulons construire des programmes solides, et le langage de programmation va nous permettre de créer nos briques. Mais il ne sert

à rien de construire sur du sable ; la compréhension de la nature et de la structure des données, la connaissance de l'algorithmique constituent les fondations de toute construction correcte. C'est un passage difficile, mais obligé. Pour cette raison, nous insistons beaucoup sur les aspects fondamentaux (et en particulier, algorithmiques) de la programmation.

Un second cours lui fait suite ([1]), tourné vers les aspects plus formels de l'algorithmique.

0.1.2 Le choix du langage

Le choix d'un langage de programmation pour une initiation à l'informatique n'est pas, comme on l'a déjà laissé entendre, entièrement anodin. L'École des Mines de Saint-Etienne, qui ne répugne à aucune expérience dans le domaine, a utilisé pour ce faire des langages aussi variés que *Fortran*, *Algol 60*, *APL*, *Pascal*, *Scheme*, *ADA*, *C*, *C++*, *Matlab*, *Scilab*, et probablement quelques autres encore. Qu'ils soient impératifs, fonctionnels, déclaratifs, à objets ou à enzymes gloutons[1], ces langages ont leurs avantages et leurs inconvénients, leurs partisans et leurs détracteurs, et ont fait parmi les étudiants des heureux, des insatisfaits, et dans tous les cas beaucoup d'indifférents.

Ce cours a choisi comme langage support le langage *C* qui est en lui-même un petit langage relativement simple à apprendre, et comportant peu de concepts. Il constitue une bonne introduction, pour le programmeur, aux langages à objets tels que *C++* et *Java*.

Malgré cette simplicité, c'est un langage largement utilisé dans le domaine industriel : on le trouve ainsi comme langage de développement du système Unix (et toutes ses variantes), et des utilitaires associés. On le trouve également dans des applications d'informatique embarquée, de contrôle de procédés industriels, de calculs scientifiques, de traitements de données diverses...

C'est enfin un langage mature, donc peu soumis à des (r)évolutions rendant les anciens programmes incompatibles, et disposant sur quasiment toutes les plates-formes d'environnements de développement performants (et du domaine public, ce qui est un avantage non négligeable).

Le langage C, bien sûr, ne présente pas que des avantages. Conçu pour permettre au programmeur d'utiliser de manière efficace la machine sous-jacente, il impose à son utilisateur :

– de connaître l'architecture générale d'un ordinateur, afin de comprendre comment ces concepts se matérialisent dans ce langage de programmation ;

[1] Terme de remplissage, utilisé ici en attendant la prochaine révolution...

– de savoir manipuler le système d'exploitation, puisque, C étant un *langage compilé*, la mise en œuvre d'un programme nécessite une séquence d'opérations non triviales pour le néophyte.

Le chapitre 1 a donc pour objectif annexe de présenter la structure d'un ordinateur (du matériel jusqu'au logiciel), et de décrire sommairement la mise en œuvre du langage C sur le système le mieux adapté : UNIX, et en l'occurence, Linux

0.2 La finalité de ce cours

Ce document a pour but de présenter des techniques d'analyse permettant de passer, par étapes successives, d'un problème, énoncé en français, à un programme résolvant ce problème, écrit dans le langage choisi pour cet apprentissage, à savoir le langage `C`.

Le lecteur averti pourra aisément transposer les concepts à tout langage du même genre que `C`, c'est-à-dire un langage de type procédural, intégrant la notion de fonctions (ou procédures, ou sous-programmes), ainsi que la possibilité de créer des structures de données : `Pascal`, `C++`, `VBA`, sont des langages ne posant pas de problème à ce niveau.

Cette présentation s'appuie largement sur des exemples, simples au départ, puis de plus en plus complexes. Il est difficile en effet de présenter d'emblée **la** méthodologie universelle (à supposer qu'il en existe une) : ceci, à notre avis, ne revêt un intérêt qu'après une certaine pratique effective de la résolution de problèmes de complexité non triviale.

Dans cette présentation sont successivement abordées différentes facettes de cette analyse, à savoir l'algorithmique, la structuration par les traitements et la structuration par les données.

Il faut bien comprendre que ce qui est présenté ne revêt pas le caractère de règles absolues : ne pas les suivre ne condamne pas nécessairement à obtenir des programmes incorrects. Cependant, les suivre permet d'obtenir des programmes plus facilement compréhensibles par quelqu'un qui ne les a pas écrits[2], et améliore donc la maintenance de ceux-ci (que ce soit pour corriger ou pour améliorer le programme). On est donc ici plutôt dans le domaine des *bonnes pratiques*.

0.3 Déroulement du cours

Le cours « Introduction à l'informatique » comporte six séances de trois heures, la dernière étant entièrement consacrée à la réalisation complète du premier programme (mini-projet).

[2] ou ne les a pas relus depuis un moment...

Ce petit support de cours recouvre les cinq séances d'introduction au langage *C*. Chaque séance comporte une phase magistrale, et une phase de travaux dirigés et pratiques.

0.3.1 Séance 1 : introduction au matériel et au logiciel

But Savoir éditer, compiler et exécuter un programme ; apprendre la syntaxe des expressions arithmétiques.

Théorie Présentation de notions sur l'informatique, les machines, les algorithmes, les environnements de développement.

Introduction au langage C : premiers éléments de syntaxe, types de données (entiers, réels, caractères) ; représentation des données ; variables ; expressions.

Pratique Maîtriser un environnement de programmation. Démonstration de l'environnement. Le célèbre programme « Salut, monde ».

0.3.2 Séance 2 : quelques problèmes élémentaires

But Comprendre les notions de variable, d'expression, et les itérations (boucles et tests)

Théorie Modélisation de problèmes.

Pratique Partir d'un problème simple, (*PGCD*, calculer un sinus), en décrire les étapes de la résolution.

0.3.3 Séance 3 : procédures

But Comprendre la notion de procédure.

Théorie Description et appel de procédure. Passage de paramètres. Prototypes des procédures.

Pratique Étapes de la création d'une application : décomposition en procédures. Procédures et compilations séparées. Utilisation de procédures de bibliothèque correspondant aux fonctions mathématiques usuelles.

0.3.4 Séance 4 : tableaux

But Les tableaux.

Théorie Opérations simples sur tableaux. Tableaux à deux dimensions, ou matrices.

Pratique Recherches de minimums, maximums, etc. Remplissage de tableau par des valeurs aléatoires. Exemple du calcul des moyennes.

0.3.5 Séance 5 : du problème au programme

But Savoir traiter un problème plus complexe.

Théorie Analyse de problème, décomposition en sous-problèmes par les données ou par les traitements.

Pratique Analyse, conception du problème traité en mini-projet.

0.3.6 Séance 6 : une application

But Séance d'intégration pour obtenir le logiciel dont la réalisation est visée dans le cadre du cours.

Pratique Présentation de quelques principes de mise au point.

0.4 Informations en ligne

De nombreuses informations (compléments de cours, exercices corrigés, références bibliographiques ou Internet, copies de supports de cours) sont disponibles sur le site Internet du cours informatique :

`http://kiwi.emse.fr/INTROINFO/`

0.5 Notes techniques

Ce support de cours a été rédigé par Jean-Jacques Girardot et Marc Roelens à partir de divers documents antérieurs.

Le document lui-même a été créé sous sous Linux, au moyen de LyX ([3]), un éditeur de document qui fournit un interface WYSIWYM avec TeX et LaTeX ([2]).

Chapitre 1

Introduction

1.1 Cours

Ce premier chapitre introduit les diverses notions de l'informatique pratiquement indispensables à la compréhension de l'activité de programmation : ce qu'est un ordinateur, un fichier, un compilateur, et comment mettre en œuvre toutes ces choses...

Après cette présentation sommaire, le cours décrit quelques éléments de base du langage C ainsi que l'utilisation de celui-ci sur les machines.

1.1.1 Modèle de l'ordinateur

1.1.1.1 Architecture d'un ordinateur

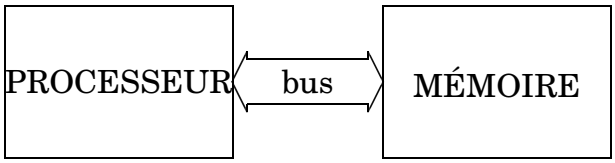

FIG. 1.1 – L'architecture de Von Neumann

Les ordinateurs actuels dérivent d'une architecture définie en 1946, la *Machine de von Neumann* (c.f. figure 1.1). Les deux composantes principales sont le *processeur* (ou *unité centrale*) et la *mémoire*, reliés entre eux par le *bus*.

Dans cette machine, le problème à résoudre est décrit sous la forme *d'instructions à exécute*r, le *programme*. L'exécution d'un programme

est dite *processus*. La mémoire de la machine contient à la fois les instructions et les données du processus. Le jeu d'instructions du processeur n'est pas universel, mais dépend du modèle de la machine.

D'autres éléments sont connectés au bus, ou à des bus auxiliaires : les *périphériques* (écran, clavier, souris, imprimante, modem, scanner, disque dur, lecteur de disquette, de cédérom, etc.), qui permettent la communication avec le monde extérieur et le stockage de l'information. Ces éléments sont commandés par des instructions spécifiques du processeur.

1.1.1.2 La mémoire

La mémoire d'un ordinateur, dite parfois *mémoire centrale*, peut être considérée comme un espace linéaire d'éléments identiques, les *cellules de mémoire*. Les éléments d'une mémoire de taille N sont désignés par leurs *index*, entiers de l'intervalle $[0,\ N-1]$. On appelle également *adresse mémoire* l'index désignant une cellule mémoire.

Dans les architectures actuelles, ces cellules sont représentées par le regroupement, dit *byte* ou *octet*, de huit indicateurs élémentaires, les *bits*, pouvant avoir chacun la valeur 0 ou 1. Un *byte* permet donc de représenter 2^8 valeurs différentes. En langage C, une telle donnée permet de répresenter :

- soit les valeurs entières comprises entre -128 et +127 (ce qui est appelé en C le type `char`);
- soit les valeurs entières comprises entre 0 et 255 (ce qui en C est appelé en C le type `unsigned char`).

Selon les besoins du programmeur (et selon les langages de programmation), les cellules de mémoire peuvent être regroupées (par 2, 3, 4, 8, 16...) pour représenter des données entières (pouvant prendre de plus grandes valeurs), mais également des données à virgule fixe ou flottante, des caractères imprimables, des chaînes de caractères...

1.1.1.3 Le processeur

Le processeur exécute des programmes situés en mémoire centrale. Ces programmes comportent des données et des instructions destinées à transformer ces données.

Une partie du processeur est *l'unité arithmétique et logique*, dont le but est d'effectuer diverses instruction de calcul.

Le processeur dispose d'un petit nombre d'emplacements de mémoire à accès très rapide (dit *registres*, au nombre de 16, 32, etc), qui lui permettent de stocker temporairement des valeurs d'éléments, et d'effectuer des opérations sur ces valeurs (le processeur ne peut effectuer

aucune opération *directement* sur le contenu d'une cellule mémoire).

1.1.1.4 Communication entre processeur et mémoire

Par l'intermédiaire du bus, le processeur peut indiquer une adresse mémoire et lire le groupe d'éléments (des octets, 2, 4, 8, 16, etc, selon les machines) situés à cette adresse. Les valeurs lues sont placées dans un ou plusieurs registres du processeur. Celui-ci est ainsi capable de *lire* les instructions et les données lorsque nécessaire.

Le processeur peut également effectuer le transfert d'une donnée contenue dans un registre vers une cellule mémoire.

1.1.1.5 Les disques durs

Les disques durs sont des unités de stockage de l'information de grande capacité (dix à cent fois la taille de la mémoire centrale), dont le rôle est d'assurer la conservation permanente des programmes et données de l'utilisateur. Ces informations sont organisées sous la forme d'un ou plusieurs *systèmes de fichiers*.

1.1.1.6 Les systèmes de fichiers

Un système de fichiers (ceci est vrai pour Unix, Linux, Windows, etc) est une organisation auto-décrite, qui comprend deux types d'objets : les *répertoires* et les *fichiers*.

Un **fichier** est une séquence d'octets, désignés globalement par un nom. Ces octets permettent de représenter des données de l'utilisateur ou du système.

Un **répertoire** contient des *désignations symboliques* (les noms) de fichiers ou d'autres répertoires. Il existe dans tout système de fichiers un répertoire particulier, qui est la *racine* de ce système de fichiers. Tout fichier est désigné par un nom référencé dans un répertoire, et un seul. Tout répertoire est désigné par un nom, référencé dans un autre répertoire, ou dans la racine. Répertoires et fichiers constituent ainsi une *arborescence*.

Une référence à un fichier particulier (l'on parle parfois de « *chemi-nom* ») se note par la liste des répertoires qu'il faut traverser, depuis la racine, pour aboutir à ce fichier. La syntaxe de cette désignation dépend des systèmes d'exploitation.

– Sous UNIX ou Linux, c'est le caractère "`/` " (*slash*) qui sert de séparateur. On écrira :

```
/users/dupont/exemples/prog17.c
```

Le système de fichiers porte ici le nom « users » ; le premier slash indique que l'ensemble a la syntaxe d'une référence absolue.

– Sous Windows, c'est le caractère "\" qui sert de séparateur ; la désignation débute par le nom du disque sur lequel se situe le système de fichiers :

```
c:\users\dupont\exemples\prog17.c
```

Un système de fichiers peut être installé sur un disque dur, mais aussi un CD-ROM, une disquette (sur laquelle l'espace disponible à limité à 1,44 millions d'octets), etc.

Fichiers et répertoires sont désignés par des noms, qui sont habituellement des suites de lettres et de chiffres. Certains systèmes sont très restrictifs, tels **Dos**, qui impose des désignations de la forme « XXXXXXXX.YYY », c'est-à-dire huit caractères, le « nom » proprement dit, suivis d'un point, suivis de trois caractères, l'« extension ». D'autres systèmes acceptent, avec certaines restrictions, des suites arbitraires de caractères (**Mac OS**, **Windows**). Sous **UNIX** ou **Linux**, il est possible d'utiliser de tels noms, à condition de les placer entre caractères « quotes ».

Nous conseillons pour notre part d'utiliser des noms relativement courts (huit lettres, ou moins), constituées de lettres et de chiffres.

Notons que l'*extension* (le *suffixe* de la désignation d'un fichier) joue un rôle important dans la compréhension par l'utilisateur de la nature des fichiers. Ainsi, les extensions ".c" et ".h" sont à réserver pour les programmes écrits en C, les fichiers ".o" représentent des modules objets, résultats de la compilation d'un module source, les ".exe" représentent des programmes exécutables, les ".txt" des fichiers de textes, les ".doc" et ".rtf" sont des fichiers créés par l'utilitaire « **Word** », etc. Dans certains cas, le système d'exploitation utilise cette extension pour savoir à quel programme associer à tel fichier.

L'on voit, par ces derniers exemples, que les fichiers répondent à de multiples finalités. De manière très générale, on peut dire qu'ils constituent un support pérène[1] à l'information. Ils permettent de représenter le systèmes, les informations de l'utilisateur, et tout un ensemble de données utilisées temporairement par le système ou les programmes de l'utilisateur : les zones de travail de éditeurs de texte et des compilateurs, les mails envoyés ou reçus, etc.

[1]Plus simplement dit, leur contenu perdure lors de la mise hors-tension de l'ordinateur, alors que celui de la mémoire centrale et des registres de la machine disparaît.

1.1.1.7 La programmation des ordinateurs

La programmation d'un ordinateur se réalise à travers le *langage machine*, qui est complexe et propre à chaque modèle d'ordinateur. L'on utilise dans la pratique des *langages de haut niveau*, tels le C, le Fortran, le Java, etc, qui sont "universels". Le plus souvent, un programme spécialisé, dit *traducteur* ou *compilateur*, transforme le programme réalisé en un programme en langage machine spécifique à l'ordinateur sur lequel on travaille : c'est le processus de *compilation*.

1.1.1.8 Le système d'exploitation

Le système d'exploitation est un complément indispensable à l'utilisation de tout ordinateur. Il a pour tâche de gérer les *ressources* de l'ordinateur (processeur, mémoire, et surtout périphériques) et de fournir un support à l'utilisateur, en contrôlant l'exécution des programmes et utilitaires (compilateur, éditeur de texte, etc.).

Le système d'exploitation est « automatiquement » chargé en mémoire centrale lors de la mise sous tension de l'ordinateur, et va dès lors gérer le fonctionnement de celui-ci.

1.1.1.9 L'environnement de travail

L'environnement de travail, enfin, est l'indispensable intermédiaire entre l'utilisateur et les ressources de la machine et du système. Il permet à l'utilisateur de déclencher l'exécution de programmes préexistant sur la machine, dont le rôle est de rendre celle-ci utilisable sous une forme conviviale, que ces programmes aient pour but de consulter le Web, lire ou écrire du courrier, créer des fichiers, des documents, ou d'autres programmes.

L'environnement de travail peut offrir un système de multi-fenêtrage (Windows, Gnome sous Linux, etc.) faisant grande consommation des interactions souris, ou un mécanisme basé sur des lignes de commandes telles que :

```
gcc -o prog.exe prog.c
```

L'une des interfaces les plus simples permet uniquement de saisir des lignes tapées au clavier ; chaque ligne est censée représenter une commande à exécuter. Nous disposerons sous *Windows* d'un interprète élémentaire (de nom « invite de commande » ou parfois « boîte MS-DOS »), et également d'un interprète plus évolué, appelé *bash*. L'on utilise souvent le terme générique de *shell* pour désigner ces interfaces.

1.1.1.10 Fonctionnement général et processus d'exécution

Parmi les registres du processeur, il en existe un, dit *compteur programme*, qui désigne à tout instant une adresse en mémoire centrale : c'est celle de la prochaine instruction à exécuter. Lors de l'exécution, le processeur effectue les opérations suivantes :

1. il va chercher, dans la mémoire centrale, l'instruction désignée par le *compteur programme*, et la place dans un registre (dit *registre instruction*) ;
2. il incrémente le *compteur programme*, afin que celui-ci désigne l'instruction suivante ;
3. il décode l'instruction, vérifiant qu'elle est correcte ;
4. il va éventuellement rechercher en mémoire centrale le, ou les opérandes désignés par l'instruction, et les place dans des registres ;
5. il exécute l'instruction ; cette instruction peut impliquer la modification de certains registres, y compris le compteur programme ; on dit alors que l'instruction est un *saut*, ou un *branchement*.
6. il va éventuellement écrire en mémoire le contenu d'un ou plusieurs registres ;
7. le cycle se continue par un retour à l'étape 1.

Le cycle d'exécution est le même, que le processeur soit en train d'exécuter des instructions du système d'exploitation, de l'environnement de l'utilisateur, ou du programme de l'utilisateur.

1.1.1.11 Notion de variable

La mémoire centrale ne contient pas que des instructions, mais sert aussi à représenter les données manipulées par le programmeur. Quand le programmeur souhaite utiliser une ou plusieurs cellules mémoire pour stocker une information utile pour le programme, il doit :

- déterminer le nombre de cellules utiles, c'est-à-dire la taille de la donnée ;
- trouver un emplacement non utilisé dans la mémoire permettant de stocker cette donnée (cellules contiguës en mémoire de la taille de la donnée à stocker) ;
- mémoriser l'emplacement (l'adresse en mémoire) de la donnée pour les manipulations à venir ;
- « réserver » cet emplacement mémoire afin d'éviter que d'autres données n'utilisent le même emplacement ;

– utiliser les bonnes instructions d'échange entre le processeur et la mémoire (transférer le bon nombre d'octets, utiliser le bon registre).

Ces opérations deviennent très fastidieuses lorsque la taille du programme (et donc, le nombre de données utilisées) augmente. La plupart des langages de programmation intègrent donc une gestion des données par l'intermédiaire d'un moyen abstrait : la plupart du temps, ce moyen abstrait est un *identificateur*, qui est tout simplement un nom choisi par le programmeur.

Dans le programme source (en langage de haut niveau), la donnée est manipulée par son nom ; c'est au moment de la traduction en langage machine que le traducteur (le compilateur dans le cas du langage C) effectuera une association entre le nom et l'emplacement mémoire réel. Toutes les instructions ultérieures qui manipulent la donnée utiliseront l'emplacement mémoire associé à la donnée. Ces associations *nom-emplacement mémoire* sont désignées par le terme de *variables* : ce nom précise que le contenu de la mémoire peut en particulier être modifié par le programme ; la *valeur* d'une variable peut donc changer au cours de l'exécution.

1.1.2 Représentation des informations

Pour représenter des données complexes, les octets sont fréquemment regroupés par deux, quatre, huit. Les conventions de représentation de ces données complexes dépendent des choix des constructeurs, mais restent en général transparentes aux programmes écrits dans des langages de haut niveau, tels que le C. Ainsi, les *valeurs entières* manipulées par les programmes sont souvent représentées par le regroupement de quatre octets, fournissant 2^{32} combinaisons différentes. Les *entiers* ainsi représentés sont habituellement ceux de l'intervalle $[-2^{31},\ 2^{31}-1]$, soit encore $[-2147483648,\ 2147483647]$. En langage C, une telle donnée est dite de type `int`.

Les entiers ne conviennent pas pour toutes les applications. Le calcul scientifique nécessite la représentation de *valeurs réelles*, pouvant être très grandes ou très petites, avec une partie décimale, etc. Les machines offrent de telles représentations (qui approximent les nombres réels par des fractions dont le dénominateur est une puissance de deux), toujours en utilisant des regroupements d'octets. Ces données sont dites `float` et `double` dans le langage C, et utilisent en général 4 et 8 octets respectivement.

Enfin, les informations le plus fréquemment manipulées par les applications sont de type *alphanumérique*. Les ordinateurs utilisent tous

pour les lettres, chiffres et certains caractères, une représentation normalisée dite code ASCII, comportant 128 éléments. La table 1.1 en

	000	001	010	011	100	101	110	111
0000	NUL	DLE	SPACE	0	@	P	‘	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	’	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

TAB. 1.1 – Le code ASCII

donne la composition et la codification ; les caractères de cette table sont représentés par des octets, le premier bit de l’octet étant 0, les trois suivants le numéro de colonne et les quatre suivants le numéro de ligne (par exemple, la lettre **A** est représentée par la suite binaire 01000001). Un mot, une phrase, un texte sont ainsi représentables par une séquence d’octets. On parle souvent de *chaîne de caractères* pour désigner de telles séquences, et les langages de haut niveau disposent de notations spécifiques pour définir de telles séquences.

Le code ASCII ne représente malheureusement qu’un sous-ensemble très limité des caractères nécessaires à l’expression écrite, et d’autres normes sont utilisées pour la représentation des langages indo-européens, du japonais, etc. Malheureusement, ces normes sont multiples, et rien ne garantit la portabilité si ces caractères sont utilisés. Nous retiendrons donc que, pour ne pas avoir de souci, il est :

- **impératif** de n’utiliser que des caractères du code ASCII pour construire notre code source (instructions, variables, procédures) ;
- **préférable** d’éviter les caractères non-ASCII (et notamment les caractères accentués) dans les commentaires de programmes, ainsi que dans les chaînes de caractères.

1.1.3 Mise en œuvre d'un programme

1.1.3.1 Qu'est-ce qu'un langage de programmation ?

Nécessaire (et ennuyeux) intermédiaire entre l'homme et la machine, l'étape de *programmation* consiste à rédiger la description d'une méthode de résolution d'un problème en une forme « compréhensible » pour la machine. Cette description implique que l'on connaisse la méthode pour résoudre le problème, l'*algorithme*, et que l'on maîtrise le *langage de programmation* choisi.

1.1.3.2 Les étapes de la vie d'un programme

Un programme[2], *toujours conçu après moultes réflexions*, va d'abord être créé par l'utilisateur sous la forme d'un *programme source*, c'est-à-dire un ou plusieurs fichiers contenant le texte du programme, écrit dans le langage de haut niveau choisi, C en l'occurrence. Ce travail est effectué au moyen d'un *éditeur de texte*. Sous *Windows*, nous utiliserons par exemple l'utilitaire *NotePad*, en sauvegardant le programme sous un nom tel que `prog.c` (le suffixe "`.c`" indique que le fichier contient un programme C). Sous *Linux*, des outils tels que *Gedit* ou *KWrite* constituent de bons choix comme outil pour débuter. L'utilisateur averti, pour sa part, optera souvent pour des logiciels tels que *vi* ou *emacs*.

Lorsque le programme est terminé, et prêt à être testé, nous allons le compiler, c'est-à-dire le transformer en un *programme objet*, qui est une représentation interne, propre à la machine, des instructions traduites par le compilateur, mêlées à diverses autres informations. Cette traduction fournit un fichier dit "objet", dont le nom va être `prog.o`. En vue d'une exécution ultérieure, il convient d'ajouter à ce fichier des *utilitaires du système*, qui permettent, entre autres fonctions, d'exécuter les lectures et écritures nécessitées par le programme. Cette seconde phase est dite *édition des liens*, et fournit un fichier représentant le *programme exécutable*, de nom `prog`. Ces deux étapes (compilation et édition des liens) peuvent s'enchaîner directement par la commande unique :

```
gcc prog.c -o prog.exe
```

Enfin, l'exécution proprement dite s'effectue en entrant simplement le nom du programme à exécuter :

```
prog
```

[2]Nous nous limitons ici au cas traditionnel des langages compilés.

(dans ce cas particulier, l'interprète de commandes va rechercher le fichier de nom `prog.exe` ou `prog.com`; les deux suffixes correspondant à deux formats d'exécutables reconnus par Windows).

Note Le compilateur C opère en fait en plusieurs phases : la compilation proprement dite est précédée d'une exécution d'un *préprocesseur*, dont le rôle sera explicité ultérieurement, et qui sert principalement à insérer automatiquement des déclarations permettant au programmeur de faire appel aux opérations d'entrées-sorties dans son programme.

1.1.4 Un premier programme C

```
#include <stdio.h>
int main(int argc, char * argv[])
{
  puts("hello world");
  return 0;
}
```

FIG. 1.2 – Premier programme C

La figure 1.2 présente un premier programme C complet. Ce programme contient les éléments suivants :

- Une première ligne, destinée au préprocesseur :

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

  La syntaxe de cette ligne doit être scrupuleusement respectée. Le caractère *dièse*, `#`, lorsqu'il est le premier caractère de la ligne, indique une ligne destinée au préprocesseur C ; la commande demande l'inclusion, dans le programme qui va être compilé, du fichier `stdio.h` qui contient des déclarations permettant de faire appel aux opérations d'entrées-sorties standards du langage C. Le suffixe `.h` indique qu'il s'agit d'un fichier d'inclusion.
- Le programme C est constitué d'instructions placées à l'intérieur de la construction, entre les accolades :

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

  Ces instructions vont être exécutée en séquence. La première :

  ```
  puts("hello world");
  ```

constitue un appel de la *procédure*[3] `puts` qui affiche sur le terminal une *chaîne de caractères*. Celle-ci est définie, dans le langage C, par la suite des caractères qui la composent, placée entre doubles apostrophes. Cette chaîne est transmise à la procédure `puts`, ce que l'on indique par les parenthèses suivant le nom de la procédure. Le point virgule termine l'instruction.
La seconde instruction :

```
return 0;
```

permet l'arrêt du programme. C'est en fait un ordre de sortie d'une procédure (dans ce cas, la procédure `main`) et de retour à l'appelant (en l'occurrence, de retour au système d'exploitation), qui permet ici à l'application de spécifier un *code de retour*, qui est une valeur numérique entière. La valeur 0 indique une terminaison normale du programme. D'autres valeurs indiquent des fins anormales. Ce code de retour peut être testé par les langages de commandes.
Insistons immédiatement sur le fait que l'instruction `return` permet de quitter une procédure et de revenir à la procédure appelante, et ce n'est que lorsqu'elle est utilisée depuis le `main` qu'elle permet d'arrêter le programme.
Une autre manière d'arrêter un programme est d'utiliser un appel de la procédure *exit*, sous la forme :

```
exit(0);
```

La procédure du système `exit` transmet au système une indication sur la terminaison du programme : le nombre `0` indique ici une terminaison jugée *normale* par le programmeur. Là également, le point-virgule termine l'instruction.

Notes :

– nous respecterons à la lettre la notation :

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

Elle permet en effet de déclarer qu'il s'agit du *programme principal* : ceci permet au moment du lancement du programme de savoir quelle est la première instruction à exécuter (ce n'est pas la première instruction qui apparaît dans le code source).
Ce programme principal communique avec son environnement au moyen des deux variables `argc` et `argv`, et que ce programme

[3]Une procédure est un ensemble d'instructions répondant à une finalité précise. Beaucoup sont prédéfinies dans le système ; nous verrons au chapitre 3 comment les définir en C.

fournira au système un *code de retour* sous la forme d'un `int`. Tout ceci sera expliqué en détail ultérieurement, mais retenons toutefois que *c'est ainsi que doit être déclaré tout programme* C *conforme.*

– nous prendrons l'habitude de terminer systématiquement nos procédures `main` par une instruction `return x;` ou `exit(x);`

L'exécution du programme sous Linux est présentée à la figure 1.3, tandis que la figure 1.4 nous montre la même opération sous Windows.

```
[P]$ gcc prog1.c -o prog1
[P]$ prog1
hello world
[P]$
```

FIG. 1.3 – Exécution du programme sous un *shell* Linux

```
C:\temp> gcc prog1.c -o prog1.exe
C:\temp> prog1.exe
hello world
C:\temp>
```

FIG. 1.4 – Exécution du programme sous un *shell* Windows

Notes :

– L'impression du message `hello world` est suivie d'un *passage à la ligne* ; celui-ci est automatiquement ajouté par la procédure `puts`.

– Sous *Windows*, l'utilisation du suffixe `.exe` est obligatoire, pour indiquer qu'un fichier représente un programme exécutable.

– Sous *Linux*, un autre mécanisme est utilisé pour repérer les programmes exécutables, et l'utilisation d'un tel suffixe n'est pas nécessaire. Cependant, pour des raisons de sécurité, *Linux* n'accepte d'exécuter que les programmes situés dans un ensemble prédéfinis de répertoires (ensemble désigné par le contenu de la variable du Shell `PATH`. S'il n'est pas déclaré que le *répertoire courant* fait partie de cet ensemble, il faut utiliser une notation telle que :

```
[P]$ ./prog1
```

pour lancer l'exécution du programme.

```
#include <stdio.h>
int main(int argc, char * argv[])
{
 float diametre=12.25;
 float pi=3.1415926535;
 float surface;
 surface = pi*(diametre/2)*(diametre/2);
 printf("La surface du disque de diamètre %f est %f\n",
   diametre, surface);
 return 0;
}
```

FIG. 1.5 – Surface d'un disque

1.1.5 Un second exemple

La figure 1.5 propose un autre exemple, calculant la surface d'un disque dont le diamètre est donné.

Nous avons donc besoin dans ce programme de représenter au moins deux données : le diamètre et la surface calculée. Ces données doivent être représentées par des nombres en virgule flottante (aucune raison particulière pour que ce soient des entiers), ce qui en C s'appelle le type `float` ; nous dénommerons ces variables `diametre` et `surface`. Nous représenterons également la valeur de π par une variable dénommée `pi`.

Ce programme contient alors les éléments suivants :

- comme dans le cas précédent, l'inclusion des déclarations nécessaires aux entrées-sorties ;
- la déclaration "habituelle" de la fonction `main` ;
- les déclarations des trois variables de type `float`, deux d'entre elles recevant une valeur initiale ; remarquons la syntaxe d'une déclaration de variable :
  ```
  nomdetype nomdevariable[=valeurinitiale];
  ```
  la valeur initiale étant optionnelle ; lorsque plusieurs variables sont de même type, on peut les déclarer par une seule instruction, et les trois déclarations pourraient être remplacées par :
  ```
  float diametre=12.25,pi=3.1415926535,surface;
  ```
  un nom de variable (ce que l'on désigne souvent par le terme d'*identificateur*) peut comporter des caractères alphabétiques (attention, **non accentués**) minuscules ou majuscules, des chiffres (sauf en première position) ou le caractère _ ;
- les instructions proprement dites :

– la première calcule la surface du disque, elle utilise les variables `diametre` et `pi` (dans la partie à droite du signe égal), et la variable `surface` (dans la partie à gauche du signe égal) ; elle met en œuvre une opération appelée *affectation*, dont la syntaxe est
variable = *expression*
Son rôle est de stocker *dans la case mémoire associée à la variable* la valeur de l'expression : le contenu de la case mémoire est ainsi modifié.
L'expression elle-même fait appel aux opérations de multiplication et division (représentées par `*` et `/` respectivement), et aux parenthèses servant à regrouper des calculs. Dans cette expression apparaissent également des noms de variables : dans ce cas, on calcule la valeur de l'expression en *remplaçant le nom de la variable par le contenu de la case mémoire qui lui est associée*.
– la seconde affiche le résultat au moyen d'une procédure du système, `printf`, qui permet l'impression de messages et de valeurs numériques.
Notons le fonctionnement de `printf`. Il y a ici trois paramètres, séparés par des virgules, le tout étant entre parenthèses.
  – le premier paramètre, une chaîne de caractères, décrit de quelle manière l'affichage doit s'effectuer ; dans cette chaîne :
    – un caractère autre que % est imprimé tel quel.
    – le caractère % introduit la description d'un affichage : `%f` indique que l'affichage correspond à un nombre de type `float`.
    – enfin, `\n` est une convention d'écriture qui permet de représenter, à l'intérieur d'une chaîne, le caractère *LF* (*linefeed*, ou *passage à la ligne*), de code ASCII `00001010`. L'impression de ce caractère provoque un passage à la ligne.
  – les autres paramètres représentent les valeurs à imprimer ; la première spécification d'affichage de la chaîne, ici `%f`, décrivant comment s'imprime la première valeur (le contenu de la case mémoire associée à la variable `diametre`), la seconde spécification (encore `%f`) s'appliquant à la seconde valeur (le contenu de la case mémoire associée à la variable `diametre`, etc.

Signalons, toujours à propos de `printf`, que `%f` fournit un affichage par défaut ; une notation telle que `%12f` indique que l'on désire que l'affichage du nombre occupe 12 caractères exactement, et `%12.4f` précise que l'on veut en outre exactement 4

Mots clefs	Description	Taille sur nos machines
`int`	entier	4 octets
`long`	entier	4 octets
`short`	entier	2 octets
`char`	entier	1 octet
`long long`	entier	8 octets
`float`	flottant	4 octets
`double`	flottant	8 octets
`long double`	flottant	12 ou 16 octets

TAB. 1.2 – Types de données du langage C

chiffres après la virgule.
- la syntaxe des lignes est relativement souple ; des blancs, ou des passages à la ligne, peuvent être insérés (sauf à l'intérieur des mots-clefs, identificateurs de variables et constantes) afin d'augmenter la lisibilité des programmes.

Le résultat affiché par le programme est :

```
La surface du disque de diamètre 12.250000 est 117.858818
```

Note importante : on a indiqué que **toutes** les déclarations de variables doivent précéder **toutes** les instructions ; même si certains compilateurs tolèrent un entrelaçage des instructions et des déclarations, cette pratique ne doit en aucun cas être recommandée !

1.1.6 Types et variables en C

1.1.6.1 Types scalaires

Le langage C utilise des mots-clefs spécifiques pour désigner les types de données de base, que l'on nomme parfois *types scalaires*, manipulés dans le langage. Le tableau 1.2 décrit ceux que nous utiliserons le plus fréquemment. Quelques commentaires sur ce tableau :

- les tailles indiquées dépendent des machines et des compilateurs. L'opération `sizeof()` permet de connaître la taille en nombre d'octets d'une donnée ou d'un type de donnée. La norme du langage précise simplement que :

$$sizeof(char) \leq sizeof(short) \leq sizeof(int) \leq sizeof(long)$$

$$sizeof(float) \leq sizeof(double) \leq sizeof(longdouble)$$

- tout type entier peut être précédé du mot-clé `unsigned`qui indique que le type doit être considéré comme « non-signé », c'est-à-dire positif ; un type signé sur p bits (8, 16, 32 ou 64) permet de coder les valeurs entières allant de -2^{p-1} à $2^{p-1}-1$, le type non-signé permet de coder les valeurs entières allant de 0 à 2^p-1 ;
- `long` et `short` sont en fait des abréviations pour `long int` et `short int`, qui sont des écritures autorisées ;
- le type `long long` est une extension non normalisée (le compilateur signale ce problème si l'on utilise les options `-ansi -pedantic -Wall`);
- les types flottants utilisent une représentation normalisée (dite *IEEE-754*) permettant de coder des nombres à virgule flottante ; le type `float` et `double` sont des types standards, le type `long double` est une extension dont la taille dépend de l'environnement de développement (12 octets par défaut sous Linux, 16 octets sous Solaris ou IRIX, 16 octets sous Linux avec option de compilation particulière) ; les précisions relatives des ces représentations (qui sont des représentations approchées des nombres réels) sont de l'ordre de 10^{-7} pour le type `float`, de 10^{-16} pour le type `double` et de 10^{-32} pour le `long double` sur 16 octets.

1.1.6.2 Constantes

Le langage C permet d'utiliser des constantes numériques dans le code source. Les formats de ces constantes sont précisés ci-dessous :

- constantes entières
  - notation décimale : succession de chiffres éventuellement précédée d'un signe
    ```
    12 -123 +345
    ```
  - notation octale : succession de chiffres commençant par `0`
    ```
    0123 077770
    ```
  - notation hexadécimale : succession de chiffres précédée des caractères `0x` ou `OX` ; les chiffres hexadécimaux sont les chiffres de `0` à `9`, ainsi que les lettres de `a` (ou `A`) à `f` (ou `F`)
    ```
    0x101 0XFADA 0xBeef
    ```
  - notation caractères : succession de caractères entre quotes `'` ; écriture en base 256, chaque caractère représentant son code ASCII ; le plus souvent limitée à un seul caractère
    ```
    'a' 'ab' '12b' '1234'
    ```
- constantes flottantes
  - un signe éventuel, une partie entière (succession de chiffres décimaux), un point `.`, une partie décimale (succession de chiffres décimaux), un exposant éventuel constitué de la lettre `e` ou de

la lettre `E`, suivi d'un signe éventuel et d'un entier décimal
`1.34 -1. +0.45 .37 1e-10 -1.23E+9 +.12E-2`

1.1.6.3 Arithmétique mixte

Le mélange de valeurs entières et flottantes (constantes, variables) est autorisé dans une expression numérique. Lorsqu'un opérateur est utilisé avec des opérandes de types différents (entier et flottant), la valeur entière est convertie en flottante avant l'application de l'opérateur. Ainsi, dans le programme de calcul de la surface d'un disque (fig. 1.5 page 23), l'expression `diametre/2` est équivalente à `diametre/2.0`. Cependant, dans un calcul tel que `i/2*f`, où `i` est un entier et `f` est un flottant, le calcul intermédiaire `i/2` s'effectue en entier, puisque les deux opérandes de la division sont des entiers, et c'est une division entière qui est exécutée. Donc, si `i` a la valeur `7` et `f` la valeur `3.5`, `i/2*f` fournit comme résultat `10.5`, tandis que `f*i/2` fournit `12.25`...

1.1.6.4 Variables et adresses

On a indiqué que le compilateur, lorsqu'il trouve une déclaration de variable, va réserver un emplacement mémoire pour stocker la donnée : c'est que l'on appelle l'adresse mémoire de cette donnée.

Le langage C offre la possibilité d'avoir accès à cette adresse : il suffit pour cela d'utiliser l'opérateur du langage & (*ampersand* en anglais, *esperluette* en français, parfois désigné par la locution « et commercial ») [4] devant le nom de la variable.

La procédure `printf` précédemment mentionnée possède un format spécifique `%p` permettant d'afficher une adresse. Ainsi, le petit programme ci-dessous :

```
#include <stdio.h>

int main(int argc, char *argv[])
{
int i = 1, j = 2;

  printf("i=%d, adresse(i)=%p\n",i,&i);
  printf("j=%d, adresse(j)=%p\n",j,&j);
  return 0;
}
```

[4] Voir l'URL `http ://www.adobe.fr/type/topics/theampersand.html`

donne à l'exécution :

```
i=1, adresse(i)=7fff2ee0
j=2, adresse(j)=7fff2ee4
```

On notera que le décalage entre les deux valeurs (affichées dans un format hexadécimal) est de 4 octets, correspondant bien à la place mémoire d'un entier.

De plus, le langage C permet également de manipuler des variables de type adresse. Pour déclarer une telle variable, on précédera le nom de la variable du caractère `*`, comme par exemple :

```
int i=2,*j;
```

qui définit une variable `i` de type `int` (de valeur initiale 2), et une variable `j` de type « adresse en mémoire d'une donnée de type `int` ».

On pourra alors mémoriser la valeur d'une adresse d'entier dans la variable `j` comme par exemple :

```
j = &i;
```

Notons que le compilateur vérifie la cohérence de type, c'est-à-dire ici que l'adresse de `i` est bien du type « adresse d'entier ».

Enfin, le langage C dispose de l'opérateur `*`, qui est l'opérateur réciproque de l'opérateur `&`, c'est-à-dire un opérateur qui, placé devant une variable de type adresse, permet de désigner le contenu de la case mémoire désignée par cette adresse. Cet opérateur est utilisable aussi bien dans une expression que dans une partie gauche d'affectation, comme dans les exemples suivants :

```
printf("*j=%d\n",*j);
*j=4;
printf("i=%d\n",i);
```

qui, à l'exécution, affiche :

```
*j=2
i=4
```

Ceci s'explique comme suit :

- `j` contient l'adresse mémoire de `i`, variable de type `int` contenant la valeur 2 ;
- la première impression de message utilise l'expression `*j`, c'est-à-dire « le contenu de la case mémoire dont l'adresse est dans la variable `j` » ; c'est donc bien la valeur de `i` (2) qui est affichée ;

- l'instruction `*j=4;` modifie le contenu de la case mémoire dont l'adresse est stockée dans la variable `j`, ce qui revient donc exactement à effectuer l'instruction `i=4;`
- la deuxième impression affiche la valeur de `i`, qui est bien 4 !

Terminons ce paragraphe par quelques remarques :

- on utilise souvent la terminologie de *pointeur* comme synonyme d'adresse mémoire ;
- l'environnement de développement garantit que l'adresse mémoire 0, souvent dénommée `NULL`, ne peut désigner une adresse mémoire contenant une donnée d'intérêt pour le programme ; cette valeur particulière sera très souvent utilisée pour signifier que la variable ne désigne aucune adresse réelle ;
- le langage C use beaucoup (abuse ?) de la notion d'adresse mémoire : il est donc relativement important de bien la comprendre.

1.2 À retenir

- La procédure `main` correspond au programme à exécuter.
- Les inclusions du préprocesseur se placent en tête du programme

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

- Toute variable doit être déclarée avant son utilisation ; elle peut recevoir une valeur initiale

```
int toto;
int titi=530;
float facteur=6.55957;
```

- Les instructions du langage :
  - expressions arithmétiques entières, flottantes, mixtes.
  - affectation : "variable" = "expression" ;
  - impression :
    - `puts("`« chaîne de caractères »`");`
    - `printf("`« format »`\n", `« valeurs »`);`

1.3 Travaux pratiques

Chaque étudiant enverra les programmes réalisés au cours de ces travaux pratiques sous la forme d'un *mél* unique, comportant les textes des programmes, séparés par trois lignes blanches les uns des autres, à l'adresse `girardot@emse.fr`. Le sujet du *mél* comportera le **nom de**

l'étudiant, et la mention TP 1.

1.3.1 Exercice 1

Tapez et testez les deux exemples de programmes donnés en cours.

Note : il n'est pas nécessaire d'envoyer ces deux programmes dans votre mél de compte-rendu.

1.3.2 Exercice 2

Écrire un programme réalisant une conversion de degrés Fahrenheit vers des degrés Celsius. La valeur à convertir est 327,35° F.

1.3.3 Exercice 3

Écrire un programme convertissant des francs en euros et réciproquement. Les résultats seront affichés avec deux chiffres significatifs, en respectant les consignes légales de conversion. Les valeurs à convertir sont 592,25 francs et 155.76 euros.

Chapitre 2

Quelques problèmes élémentaires

2.1 Cours

Ce chapitre s'intéresse plus spécifiquement à la modélisation de problèmes simples, et aux étapes de l'analyse permettant la transformation de ces problèmes en programmes pouvant être traités par l'ordinateur. Les premiers exemples sont volontairement pris dans un domaine bien connu qui est celui des mathématiques, afin de pouvoir mettre l'accent sur la méthodologie de passage de ces problèmes bien connus à l'implémentation informatique d'un programme résolvant ce problème.

2.1.1 Un premier problème : l'algorithme d'Euclide

On se propose d'écrire un programme permettant le calcul du *PGCD* de deux nombres.

2.1.1.1 L'algorithme

On se rappelle que ce problème peut être résolu (mathématiquement parlant) par l'algorithme d'Euclide. Le principe de celui-ci (qui s'applique à des entiers positifs) est le suivant : retrancher du plus grand des deux nombres le plus petit. Quand un nombre devient nul (ou, autre test équivalent, devient égal à l'autre), la valeur de l'autre nombre est le *PGCD* des deux nombres initiaux[1].

Nous nous intéresserons ici au problème de la transcription en C de cet algorithme. Nous noterons ces 2 nombres A et B, en convenant que

[1] Ceci est la version de l'algorithme d'Euclide qui n'effectue que des soustractions, il en existe une autre qui a besoin de l'opérateur modulo

nous maintiendrons toujours $A \geq B$.

Les étapes de l'algorithme sont les suivantes :

1. vérifier que $A \geq B$; sinon, permuter les deux nombres.
2. est-ce que $A = B$? Si oui, le *PGCD* est A et le programme est terminé.
3. sinon, retrancher B de A.
4. revenir à l'étape 1.

2.1.1.2 Passage au programme

Les deux nombres sont représentés par deux variables, notées `A` et `B` de type `int`. Détaillons maintenant les étapes décrites ci-dessus :

étape 1 Comparer deux nombres s'effectue au moyen d'un opérateur de comparaison ; les notations `<`, `<=`, `==`, `>`, `>=` et `!=` représentent respectivement les opérations de comparaison *inférieur*, *inférieur ou égal*, *égal*, *supérieur*, *supérieur ou égal*, et enfin *différent*. Ces opérations comparent leurs opérandes, et fournissent un résultat dit *booléen*, dont les valeurs, *vrai*s ou *faux*, sont représentées par les entiers 1 et 0 respectivement[2]. Elles peuvent s'utiliser dans les instructions conditionnelles, dont les syntaxes sont :

- `if (`*test*`)` *instruction-1*
- `if (`*test*`)` *instruction-1* `else` *instruction-2*

Dans le premier cas, *instruction-1* est exécutée si et seulement si *test* est la valeur *vrai*. Dans le second cas, si *test* est *vrai*, *instruction-1* est exécutée, et elle seule ; si *test* est faux, *instruction-2* est exécutée, et elle seule.

La comparaison de A et B va donc pouvoir s'utiliser dans une instruction débutant par :

```
if (A<B) ...
```

Se pose maintenant la question de la permutation des A et B. La meilleure solution consiste à utiliser une variable intermédiaire (appelons la C), sous la forme :

```
C=A ; A=B ; B=C ;
```

Cependant, cet échange se compose de trois instructions, alors que la syntaxe des instructions conditionnelles nous parle d'une *instruction unique*. La solution à ce problème consiste à regrouper ces instructions

[2]Le langage C considère en réalité que toute valeur non nulle représente la valeur *vrai* et que la valeur 0 (sous la forme d'un *int*, *char* ou *float*) est la valeur *faux*.

en un *bloc*, en les plaçant entre accolades. Le résultat constitue une *instruction composée* unique. Notre test devient ainsi :

```
if (A<B) { C=A; A=B; B=C; }
```

étape 2 Nous disposons de tous les éléments nécessaires à la transcription de cette étape, qui met en jeu une instruction conditionnelle, une impression, et un arrêt de l'exécution :

```
if (A==B)
{
  printf("La valeur du PGCD est %d\n", A);
  exit(0);
}
```

étape 3 Pas de difficulté non plus pour ce fragment de l'algorithme, qui s'écrit :

```
A=A-B;
```

étape 4 Une solution pour revenir à la première étape du programme consiste à enclore l'ensemble des instructions de celui-ci dans une forme répétitive ; voici trois syntaxes au choix :

- `for (;;)` *instruction*
- `while(1)` *instruction*
- `do` *instruction* `while(1);`

On notera que le `1` suivant les `while` représente en fait la *valeur booléenne vrai* : l'expression testée qui conditionne la répétition de la boucle est donc toujours vraie.

Là encore, nous utiliserons des accolades pour transformer notre séquence de trois étapes en une instruction composée unique :

```
for (;;)
{
  if (A<B)
  {
    C=A; A=B; B=C;
  }
  if (A==B)
  {
    printf("La valeur du PGCD est %d\n", A);
    exit(0);
  }
  A=A-B;
}
```

2.1.1.3 Le programme final

Nous pouvons maintenant proposer la version finale du programme, dont le texte source est donné dans la figure 2.1.

```
#include <stdio.h>
int main(int argc, char * argv[])
{
  int A, B, C;
  A = 834389;
  B = 944353;
  for (;;)
  {
    if (A<B)
    {
      C=A; A=B; B=C;
    }
    if (A==B)
    {
      printf("La valeur du PGCD est %d\n", A);
      return 0;
    }
    A=A-B;
  }
}
```

FIG. 2.1 – PGCD de deux nombres

L'exécution fournit la réponse suivante :

```
La valeur du PGCD est 743
```

2.1.1.4 Un cas non prévu !

On n'a pas démontré que l'algorithme ci-dessus se termine en un temps fini (démonstration laissée au lecteur). Ceci est toutefois l'occasion de préciser que, si l'un des entiers est nul alors que l'autre ne l'est pas, l'algorithme « boucle » indéfiniment.

On pourrait traiter ce problème en modifiant le test d'arrêt, par exemple en testant si la valeur de B n'est pas nulle :

```
if ((A==B) || (B==0))
{
 ...
}
```

Noter au passage que les opérateurs *logiques* en `C` s'écrivent && (et logique),, et || (ou logique), ; ne pas oublier le doublement du symbole, car les opérateurs & et | existent également, et ont un tout autre sens !

En fait, il conviendrait, au début du programme, de vérifier que les valeurs `A` et `B` satisfont les conditions d'application de l'algorithme, par exemple en insérant le fragment :

```
if (A<=0 || B<=0)
{
  printf("Algorithme non applicable\n");
  exit(1);
}
```

2.1.2 Somme des premiers entiers

Dans notre exemple précédent, le problème posé était en lui-même une description algorithmique de sa solution, description qu'il convenait simplement de traduire dans le langage C. Dans la grande majorité des cas, il convient d'abord de choisir (ou de concevoir) un algorithme adapté au problème, avant de passer à la forme informatique. Prenons comme exemple de problème «trouver, pour un N donné, la somme des N premiers entiers.»

Première version Une première approche peut consister à bâtir une boucle qui va énumérer ces nombres et calculer leur somme. Un tel programme s'écrit aisément :

```
int N, S, i;
...
S = 0;
for (i=1; i<=N; i++)
    S = S+i;
```

Notes : Cet exemple nous permet de présenter de manière informelle la syntaxe d'une boucle `for` :

for (*initialisation* ; *continuation* ; *gestion*) *instruction*

Dans cette syntaxe, la partie *initialisation* permet d'affecter des valeurs initiales aux variables qui vont être utilisées au sein de la boucle. L'expression *continuation* est un calcul d'une *valeur booléenne* qui indique si la boucle se poursuit ou s'interrompt. Enfin, la partie *gestion* permet la *modification* des variables utilisées dans la boucle avant un nouveau test. Il n'y a pas nécessairement un lien direct entre le test de

continuation de la boucle et les variables modifiées dans l'expression de *gestion* de la boucle.

Par ailleurs, l'expression ci-dessus `i++` fait appel à l'opérateur de post-incrémentation, `++`, spécificité du langage *C* (et de ses descendants, tels *C++* ou *Java*). L'opérateur `++` permet d'incrémenter une variable entière (c'est-à-dire de lui ajouter 1). Les instructions :

```
i++;
++i;
i=i+1;
```

sont équivalentes et ont pour effet d'ajouter 1 à la variable `i`. Cependant, les expressions `i++` et `++i` ont des sens différents :

- `i++` est une expression qui fournit la valeur de `i` **avant** l'incrémentation ;
- `++i` est une expression qui fournit la valeur de `i` **après** l'incrémentation.

Ainsi, après l'exécution du fragment de programme suivant :

```
int i, j, k; ...
i=4; j=++i; k=i++;
```

les variables `i`, `j` et `k` ont pour valeurs respectives 6, 5 et 5.

Signalons enfin l'existence de l'opérateur `-` de *décrémentation*, qui à l'instar de `++`, permet de retrancher 1 à une variable.

Seconde version Cependant, le problème de la sommation des N premiers entiers nous est connu depuis longtemps, et nous savons que :

$$\sum_{i=1}^{N} i = \frac{N(N+1)}{2}$$

Ceci nous permet de proposer la formulation suivante :

```
int N, S;
...
S = N * (N+1) / 2;
```

Cette écriture est non seulement plus élégante que notre première version, mais elle est aussi plus efficace ! Elle est, pour $N = 100$, environ 40 fois plus rapide. Mieux encore, alors que le temps d'exécution de la première version est clairement propotionnel à N (plus N est grand, plus le programme est lent), le temps d'exécution de la seconde version est *constant*, indépendant de la valeur de N.

Cet exemple trivial fait apparaitre la nécessité de trouver l'algorithme qui sera le plus efficace avant de coder cet algorithme. Nous

reviendrons naturellement sur cet aspect important de l'informatique dans la suite de ce cours, ainsi que dans le cours consacré aux algorithmes et structures de données ([1]).

2.1.3 Le calcul d'un sinus

Nous allons maintenant détailler une démarche similaire, mais plus complexe, à la faveur du problème suivant : «on cherche à calculer pour un angle donné la valeur du sinus de cet angle.»

2.1.3.1 L'algorithme

On notera x la valeur dont on cherche à calculer le sinus ; en `C`, c'est une variable de type `double`. On sait que l'on peut obtenir la valeur de *sinus(x)* par la relation :

$$\sin(x) = \sum_{i=0}^{\infty} (-1)^i \frac{x^{2i+1}}{(2i+1)!}$$

Ceci ne donne pas à proprement parler un algorithme, puisque par définition, un algorithme doit se terminer après un nombre fini d'opérations ! Mais de toutes façons, on ne cherche en fait qu'à calculer une valeur *approchée* du sinus : comment pourrait-il en être autrement avec des machines qui ont une précision limitée, et qui commettent des approximations pour représenter les nombres.

On cherche donc à majorer l'erreur commise en calculant la valeur du sinus. Pour cela, on peut remarquer que la série est alternée (change de signe, valeur absolue décroissante) pour $i > \frac{|x|}{2}$, et on sait que l'erreur commise en tronquant une série alternée au rang N (la différence entre la limite mathématique et la somme partielle au rang N) est inférieure à la valeur absolue du terme de rang N.

Si ε est la précision demandée, on obtient donc comme « formule » de calcul :

$$\begin{aligned} \sin(x) &= \sum_{i=0}^{N} (-1)^i \frac{x^{2i+1}}{(2i+1)!} \\ N &= \min(n \in \mathbb{N} \quad | \quad n > \frac{|x|}{2} \quad \textbf{et} \quad \frac{|x|^{2n+1}}{(2n+1)!} < \varepsilon) \end{aligned}$$

2.1.3.2 Détail de l'algorithme

La formule obtenue ci-dessus n'est cependant pas satisfaisante : il n'y a pas d'opérateur d'exponentiation ou de factorielle en `C`. Il faut donc continuer l'analyse.

On peut remarquer que la somme recherchée peut s'exprimer par :

$$\begin{aligned} \sin(x) = U_N &= \sum_{i=0}^{N} u_i \\ u_i &= (-1)^i \frac{x^{2i+1}}{(2i+1)!} \end{aligned}$$

et la suite u_i pouvant elle-même s'exprimer par :

$$\begin{aligned} u_0 &= x \\ u_{i+1} &= -u_i * \frac{x^2}{(2i+2)(2i+3)} \end{aligned}$$

En posant $w_i = 2i(2i+1) = 4i^2 + 2i$, on obtient :

$$\begin{aligned} u_{i+1} &= -u_i * \frac{x^2}{w_{i+1}} \\ w_{i+1} &= w_i + 8i + 6 \\ w_{i+1} &= w_i + z_{i+1} \end{aligned}$$

en posant $z_i = 8i - 2$. La valeur de z_i peut se calculer par la récurrence :

$$\begin{aligned} z_0 &= -2 \\ z_{i+1} &= z_i + 8 \end{aligned}$$

En regroupant tout cela, on obtient cette fois l'algorithme de calcul :

$$\begin{aligned} z_0 &= -2 \\ w_0 &= 0 \\ u_0 &= x \\ U_0 &= x \\ z_{i+1} &= z_i + 8 \\ w_{i+1} &= w_i + z_{i+1} \\ u_{i+1} &= -u_i * \frac{x^2}{w_{i+1}} \\ U_{i+1} &= U_i + u_{i+1} \end{aligned}$$

Comme d'habitude, on va utiliser pour chacune des suites définies ci-dessus une seule variable : la valeur de la variable à l'itération i représente la i-ème valeur de la suite. Il est donc primordial de respecter l'ordre de calcul indiqué !

Rappelons encore que le test d'arrêt consiste à tester si la valeur de $|u_{i+1}|$ est inférieure à la précision demandée (sans oublier de tester si

$i > \frac{|x|}{2}$, puisque ce n'est que lorsque cette condition est remplie que la série est alternée).

Le programme complet est donné ci-dessous :

```
#include <stdio.h>
#include <math.h>
int main (int argc,char *argv[])
{
  double x,x2,z,w,u,U;
  double epsilon;
  int i,imin;
  /* on calcule sin(pi/4) a 10-6 pres */
  x=M_PI/4;epsilon=0.000001;
  /* initialisations */
  x2=x*x;z=-2;w=0;u=x;U=x;imin=fabs(x/2);
  for (i=0;;i++) {
    z=z+8;w=w+z;u=-u*x2/w;U=U+u;
    if ((i > imin) && (fabs(u) < epsilon)) {
      (void) printf("sin(%.10g)=%.10g\n",x,U);
      (void) printf("  [apres %d iterations]\n",i+1);
      return 0;
    }
  }
}
```

Notons :

- l'utilisation de la fonction de la bibliothèque mathématique `fabs`, qui fournit la valeur absolue de son argument (argument et résultat de type `double`) ; l'utilisation de cette fonction nécessite l'inclusion du fichier d'en-tête de cette bibliothèque, nommé `math.h` ;
- l'utilisation de la *macro* `M_PI` (valeur de π) également définie dans ce fichier d'en-tête.

L'exécution de ce programme donne le résultat :

```
sin(0.7853981634)=0.7071067812
  [apres 5 iterations]
```

ce qui est un résultat tout à fait acceptable (à comparer au résultat théorique $\frac{\sqrt{2}}{2}$ soit autour de 0.707106781187). On peut noter aussi que ce résultat est obtenu assez rapidement (5 itérations représentent une trentaine d'opérations).

On peut constater sur cet exemple qu'on obtient un code très compact, mais que la lecture du code ne permet que très difficilement de retrouver ce que ce code calcule ! D'où l'utilité de documenter son code,

sans oublier de préciser, dès que l'ingénierie inverse devient difficile, la fonction précise du code en question.

2.1.3.3 Petit problème !

On veut calculer le sinus de 100. On modifie donc le programme précédent (en insérant l'instruction `x=100;` en lieu et place de `x=M_PI/4;`), et on obtient le résultat suivant :

```
sin(100)=-8.261375665e+25
  [apres 142 iterations]
```

ce qui est plutôt surprenant pour une valeur censée être comprise entre -1 et 1 !

Comme cela arrive fréquemment, l'imprécision numérique des ordinateurs est la cause du problème. La « formule » utilisée est mathématiquement exacte : cependant, elle ajoute des termes de valeurs absolues très différentes. Si le dernier terme est inférieur à la précision demandée, le terme de valeur absolue maximale, est (pour $x = 100$) le terme d'indice 49, soit :

$$|u_{49}| = \frac{100^{99}}{99!} \approx 10^{42}$$

La précision relative de la représentation des nombres flottants selon la norme IEEE-754 est de l'ordre de 10^{-16} (2^{-53} pour être précis !) : sur une valeur absolue de l'ordre de 10^{42}, on obtient donc une erreur absolue de l'ordre de 10^{26}, c'est bien l'ordre du résultat obtenu !

Dans le cas particulier de la fonction sinus, on peut améliorer grandement les choses en utilisant une propriété mathématique supplémentaire, à savoir la périodicité : se ramener à une valeur de x entre $-\pi$ et π permet d'obtenir une précision tout à fait acceptable.

On peut ainsi insérer dans les initialisations l'instruction :

```
x=fmod(x,2*M_PI);
```

en utilisant la fonction `fmod` (calcul de modulo sur des données de type `double`), appartenant toujours à la bibliothèque mathématique. L'exécution du programme ainsi modifié donne alors :

```
sin(100)=-0.5063656423
  [apres 13 iterations]
```

beaucoup plus « réaliste », et en même temps plus rapide ! On pourrait encore améliorer le programme en utilisant d'autres propriétés de la fonction sinus ($\sin(\pi + x) = -\sin(x)$ ou $\sin(\pi - x) = \sin(x)$) afin de se ramener à une valeur de x comprise entre 0 et π.

Ce qu'il faut retenir : il faut être parfaitement conscient, lorsque l'on programme une fonction, des **limitations** liées à la *programmation* de cette fonction ! Ne pas en tenir compte peut amener des conséquences non négligeables, les personnes intéressées par ce sujet pourront consulter avec intérêt la page suivante :

`http://www.math.psu.edu/dna/disasters`

pour se convaincre du caractère parfois dramatique de tels oublis :

- échec d'interception d'un missile *Scud* par un missile *Patriot* dû à une erreur de précision sur le calcul du temps ;
- échec du tir inaugural d'*Ariane V* dû à la réutilisation d'un code incorrect issu d'*Ariane IV* (et inutile dans le cas d'*Ariane V*) ;
- naufrage de la plate-forme pétrolière *Sleipner A* dû à la rupture d'un élément par mauvaise approximation d'une loi d'élasticité.

2.2 À retenir

- instruction composée
  { *séquence d'instructions* }
- opérateurs de comparaison
  `<` `<=` `==` `>=` `>` `!=`
- opérateurs logiques
  «et logique» &&, «ou logique» `||`
- boucle «infinie» :
  - `for ( ; ; )` *instruction*
  - `while (1)` *instruction*
  - `do` *instruction* `while (1);`
- boucle «for» :
  - `for (`*initialisation* `;` *continuation* `;` *gestion* `)` *instruction*

2.3 Travaux pratiques

Chaque étudiant enverra les programmes réalisés au cours de ces travaux pratiques sous la forme d'un *mél* unique, comportant les textes des programmes, séparés par trois lignes blanches les uns des autres, à l'adresse `girardot@emse.fr`. Le sujet du *mél* comportera le **nom de l'étudiant**, et la mention TP 2.

2.3.1 Exercice 1

On veut déterminer si un nombre est premier. Le programme saisit au clavier le nombre dont on teste la primalité. Si le nombre n'est pas premier, le programme donne une décomposition en deux termes non triviaux.

2.3.2 Exercice 2

Modifier le programme précédent pour imprimer tous les facteurs du nombre si celui-ci n'est pas premier.

2.3.3 Exercice 3

On veut déterminer si un nombre est parfait, c'est à dire égal à la somme de ses diviseurs, hormis lui-même. Les deux premiers nombres parfaits sont 6 (1+2+3) et 28 (1+2+4+7+14).

2.3.4 Exercice 4

Un banquier propose un placement sur le principe suivant :
- l'investissement initial est $e - 1$ euros (e est la base des logarithmes néperiens) ;
- la première année, on prélève un euro de frais de gestion ;
- la seconde année, le capital restant est multiplié par deux, et on prélève toujours un euro de frais de gestion ;
- la n-ième année, le capital restant est multiplié par n, et on prélève toujours un euro de frais de gestion ;
- le placement est prévu pour une durée de 25 ans.

Déterminer le capital restant au bout des 25 ans.

Notes :
- le fichier d'en-tête `math.h` définit la macro `M_E` ;
- essayer plusieurs types flottants ; en C, sont disponibles les types `float` (32 bits), `double` (64 bits) et `long double` (128 bits) ;
- quel est le résultat mathématique ?

Chapitre 3

Procédures

3.1 Cours

Ce cours s'intéresse plus spécifiquement à l'introduction de la notion de *procédure*, qui nous permettra de structurer et organiser efficacement le code d'un programme.

3.1.1 Retour au PGCD

Nous avions écrit lors d'une précédente séance le calcul du *PGCD* de deux nombres par l'algorithme d'Euclide, dont le texte source est rappelé dans la figure 3.1.

Que faire si notre souhait est de calculer le *PGCD* de *trois* nombres A, B et C ? On peut naturellement calculer le *PGCD* des deux premiers, A et B, puis ayant obtenu le résultat dans la variable A, recopier les mêmes instructions, en remplaçant B par C, pour obtenir le résultat désiré. La réalisation est lourde, et source de nombreuses erreurs car on oublie fréquemment de modifier un nom de variable dans le code recopié ! Et que dire s'il s'agit de calculer le *PGCD* de dix nombres, ou encore de calculer le *PGCD* à divers endroits d'un programme !

Il existe naturellement de meilleures solutions à ce problème, qui passent par l'écriture de *procédures*, dites parfois *sous-programmes*, *subroutines* (anglicisme !), et de manière plus impropre encore, *fonctions*.

Une procédure permet d'écrire une seule fois un traitement particulier, utilisé à plusieurs reprises au sein d'un programme. On verra que cette notion permet également de structurer un programme complexe, en décomposant le traitement en plusieurs sous-tâches, chacune étant représentée par une procédure (ou un ensemble de procédures).

```
#include <stdio.h>
int A, B, C;
int main(int argc, char * argv[])
{
  A = 834389;
  B = 944353;
  for (;;)
  {
    if (A<B)
    {
      C=A; A=B; B=C;
    }
    if (A==B)
    {
      printf("La valeur du PGCD est %d\n", A);
      return 0;
    }
    A=A-B;
  }
}
```

FIG. 3.1 – PGCD de deux nombres (version 1)

3.1.2 La notion de procédure

Une *procédure* est l'association d'un *nom* et d'un *ensemble d'instructions* du langage. On peut ainsi établir un certain parallèle entre procédure et instructions d'une part, variable et données d'autre part.

Définir une procédure, c'est lui donner un nom et définir son contenu, c'est-à-dire la succession des instructions qui la composent. Une fois la procédure définie, elle devient utilisable à tout endroit du programme en faisant simplement référence à son nom. En langage C, ceci se fait en accolant au nom de la procédure les caractères `()`.

Il faut bien comprendre le schéma d'exécution particulier lors de l'appel d'une procédure, qui ne suit pas le schéma d'exécution séquentiel normal du processeur : ce sont d'ailleurs des instructions spécifiques du processeur qui sont utilisées pour la gestion des procédures.

- lors le processeur exécute l'instruction spécifique correspondant à l'appel de procédure, il mémorise le compteur programme qui désigne alors l'instruction suivante (dans le flot initial d'instructions);
- le compteur programme est modifié de façon à ce que la prochaine instruction exécutée soit la première instruction de la procédure

appelée ;
– en fin de procédure, une autre instruction spécifique permet d'indiquer que le compteur programme doit être restauré avec la valeur mémorisée avant l'appel de la procédure.

Ce mécanisme particulier peut être répété autant de fois que souhaité, et il est possible que la procédure appelle une autre procédure, qui appelle une autre procédure... : le processeur mémorise à chaque fois l'adresse de l'instruction suivante, et c'est toujours la dernière adresse mémorisée qui est restaurée lors d'une fin de procédure.

On verra ultérieurement qu'une procédure peut recevoir des paramètres permettant de modifier son comportement, et fournir un résultat. Les procédures peuvent ainsi modéliser certaines fonctions mathématiques (partiellement dans tous les cas, ne serait-ce que parce que les ordinateurs ne permettent de représenter que des sous-ensembles des entiers et des réels), et parfois en calculer les valeurs.

3.1.3 La procédure *PGCD*, première version

Voici une première version de la procédure *PGCD* :

```
void pgcd(void)
{
  for (;;)
  {
    if (A<B)
    {
      C=A; A=B; B=C;
    }
    if (A==B)
    {
      return;
    }
    A=A-B;
  }
}
```

Elle reprend les instructions de calculs de notre première version du programme, en les englobant dans une déclaration de procédure :

```
void pgcd(void)
```

Cette déclaration débute par le mot-clef `void`, qui indique que la procédure ne rend pas de résultat. Le nom de la procédure (qui doit avoir les mêmes caractéristiques qu'un nom de variable) est suivi de la liste

de ses paramètres : ici, le mot-clef `void` indique qu'il n'y a pas de paramètres.

Le couple *d'accolade*s qui encadre les instructions définit le *corps* de la procédure. Enfin, on a retiré de la procédure l'ordre d'impression ainsi que l'ordre d'arrêt du programme. Ces deux instructions ont été remplacées par :

```
return;
```

qui est l'instruction en langage C permettant d'indiquer la fin de la procédure (lorsque le traitement doit s'arrêter avant la fin normale, c'est-à-dire l'exécution de la dernière instruction). Le nom de cette instruction permet d'ailleurs de mémoriser facilement que l'on *retourne* à l'exécution du programme principal.

Cette procédure va être *appelée* par l'expression suivante :

```
pgcd();
```

Notons les parenthèses, obligatoires, qui indiquent l'appel de procédure. Il n'y a rien à l'intérieur de ces parenthèses (la procédure n'a pas de paramètres), et elle ne fournit pas de résultat. Lorsque cette expression est exécutée, le code situé à l'intérieur de la procédure est exécuté.

Nous avons maintenant un autre problème à régler : les variables `A`, `B` et `C` sont définies dans la procédure `main`. Or, la procédure `pgcd` doit utiliser ces variables : mais le langage C ne permet pas à une procédure d'utiliser des variables définies dans d'autres procédures !

On peut contourner *dans un premier temps* cette difficulté en rendant les variables *globales*, c'est-à-dire connues par toutes les procédures. En langage C, ceci se fait en déclarant les variables **en dehors** de toute procédure. On peut par exemple les définir en tête du programme, et toutes les procédures définies ultérieurement dans le code source pourront y accéder (la règle générale est qu'une variable doit être déclarée avant d'être utilisée).

Notons toutefois que la variable `C` n'est utile que pour échanger les variables `A` et `B` (elle ne sert pas dans la procédure `main`) : on a donc tout intérêt à déclarer cette variable dans la procédure `pgcd`.

La figure 3.2 donne la nouvelle version, codée selon ces considérations, du programme.

3.1.4 La procédure PGCD, deuxième version

La procédure que nous venons d'écrire communique avec le programme principal (la procédure `main`) par l'intermédiaire des variables

```
#include <stdio.h>
int A, B;
void pgcd(void)
{
 int C;
 for (;;)
  {
    if (A<B)
    {
      C=A; A=B; B=C;
    }
    if (A==B)
    {
      return;
    }
    A=A-B;
  }
}
int main(int argc, char * argv[])
{
 A = 834389;
 B = 944353;
 pgcd();
 printf("La valeur du PGCD est %d\n", A);
 return 0;
}
```

FIG. 3.2 – Procédure PGCD (première version)

A et B, qui sont ici partagées entre les deux procédures (variables globales). Le programme principal positionne les deux valeurs A et B, puis appelle la procédure qui consulte (et modifie) ces deux variables, fournissant dans l'une d'elles (les deux, d'ailleurs) le résultat. Au retour de la procédure, la programme principal retrouve donc ce résultat et peut l'imprimer.

Cette approche impose au programmeur d'être parfaitement conscient du rôle de la procédure et des variables qu'elle consulte et modifie, tâche qui devient très complexe lorsqu'un grand nombre de procédures sont utilisées dans l'application.

Pour simplifier cette gestion, les langages de programmation en général, et C en particulier, permettent de déclarer qu'une procédure reçoit des *paramètres* et fournit un *résultat*. Au niveau de la déclaration, on doit alors indiquer entre les parenthèses suivant le nom de la procé-

dure la liste des paramètres (nom et type), et indiquer (le cas échéant) le type du résultat (entier `int`, flottant `double`...). La déclaration de notre procédure `pgcd` devient alors :

```
int pgcd(int X, int Y)
```

Nous déclarons cette fois que la procédure rend un résultat de type `int`, et admet deux paramètres de type `int`, connus sous les noms de `X` et `Y` à l'intérieur du code. Les noms choisis pour désigner les paramètres de la procédure n'ont de sens qu'à l'intérieur de celle-ci, et sont indépendants des noms choisis dans le reste du programme.

Une procédure est ainsi plus proche, dans son aspect, d'une fonction mathématique, si son exécution ne dépend que des seuls paramètres.

Il convient également modifier le corps de la procédure, pour que celle-ci travaille sur les variables `X` et `Y`, ainsi que l'instruction permettant la sortie de la procédure. Celle-ci rendant un résultat, nous le fournissons dans l'instruction `return`, qui s'écrit maintenant :

```
return X;
```

Enfin, le mode d'appel de la procédure est modifié. Nous écrirons par exemple :

```
R = pgcd(A,B);
```

la variable `R` ayant été déclarée de type entier.

Comment fonctionne maintenant la procédure `pgcd` ? Détaillons les étapes successives de l'exécution.

- avant l'appel, les paramètres sont calculés : ici, il s'agit juste de récuperer les valeurs contenues dans les variables `A` et `B`, mais il pourrait être nécessaire d'effectuer des opérations ; ces valeurs sont mémorisées temporairement ;
- comme avant tout appel de procédure, le compteur programme est mémorisé, puis modifié avec l'adresse de la première instruction de la procédure appelée ;
- avant même cette première instruction (ou plus exactement, lors de la première instruction de la procédure), les valeurs mémorisées pour les paramètres sont recopiés dans les variables locales (ici, `X` et `Y`) ;
- lors de l'exécution de l'instruction `return`, on calcule la valeur du résultat (l'expression après le `return`, ici simplement le contenu de la variable `X` mais qui pourrait être plus complexe), puis on restaure le compteur programme ;
- avant d'exécuter l'instruction suivante, on récupère la valeur mémorisée, ici, cette valeur est stockée dans la variable `R`. Notons que si cette valeur n'est pas immédiatement utilisée, elle est perdue !

Comme ce sont les valeurs des paramètres qui sont passées à la procédure appelée, on dit que le langage C effectue un *passage des arguments par valeur*. D'autres langages peuvent utiliser d'autres techniques (passage par adresse, par référence).

La figure 3.3 donne la nouvelle version du programme.

```
#include <stdio.h>
int pgcd(int X, int Y)
{
 int C;
 for (;;)
 {
   if (X<Y)
   {
     C=X; X=Y; Y=C;
   }
   if (X==Y)
   {
     return X;
   }
   X=X-Y;
 }
}
int main(int argc, char * argv[])
{
 int A, B, R;
 A = 834389;
 B = 944353;
 R = pgcd(A,B);
 printf("La valeur du PGCD est %d\n", R);
 return 0;
}
```

FIG. 3.3 – Procédure PGCD de deux nombres (deuxième version)

Questions :

1. Les variables `A` et `B` déclarées dans la procédure `main` sont-elles modifiées lors de l'appel de la procédure `pgcd` ?
2. Qu'en est-il si l'on appelle `A` et `B` (au lieu de `X` et `Y`) les paramètres de la procédure `pgcd` ?

3.1.5 Quelques aspects des procédures du langage C

3.1.5.1 Variables locales et globales

Il aurait été possible de nommer `A` et `B` (plutôt que `X` et `Y`) les deux paramètres de la fonction `pgcd`. Nous aurions eu alors dans notre programme coexistence de variables de même nom, mais représentant des données **différentes**. Dans le corps de la procédure `pgcd`, toute référence à la variable `A` désigne en fait « la variable `A` de la procédure `pgcd` ». C'est pourquoi on parle de *variables locales* dans ce cas d'utilisation. Il est **particulièrement recommandé** que toute variable spécifique à une procédure, qui n'a besoin d'être connue que de cette procédure, soit déclarée en tant que variable locale à cette procédure !

Cette vision présente l'énorme avantage de l'indépendance totale entre les noms (et les contenus) de variables de procédures différentes : on peut utiliser un nom de variable au sein d'une procédure sans crainte de modification du fonctionnement d'une autre procédure. Sans ce mécanisme, la gestion des noms de variables sans risque d'interférence deviendrait extrêmement lourde, donc en pratique, irréalisable !

Rappelons que le langage C, comme de nombreux langages, autorise toutefois l'utilisation de variables globales, et il faut alors prendre garde à ces risques d'interférences. Il convient donc **d'éviter au maximum l'utilisation de variables globales**, celles-ci pouvant être sources de nombreux problèmes et bogues, extrêmement difficiles à détecter et corriger.

Signalons enfin que le langage C autorise également la définition de variables locales à l'intérieur de n'importe quel bloc d'instructions (suite d'instructions entre accolades). Prenons l'exemple de la procédure `pgcd` : la variable locale `C` n'est utilisée que dans le premier bloc d'instructions, consistant à échanger les deux variables `X` et `Y`. Il aurait été possible d'écrire ce bloc d'instructions comme suit :

```
if (X<Y)
{
  int C;
  C=X; X=Y; Y=C;
}
```

Le déplacement de la déclaration « `int C;` » après l'accolade ouvrante qui suit le test « `if (X<Y)` » a pour effet de restreindre la visibité de cette variable (on utilise souvent le terme de *portée de la variable*) au

seul segment de code situé entre cette accolade ouvrante et l'accolade fermante correspondante.

Rappelons, en insistant lourdement, que la déclaration de variables locales dans une procédure ou dans un bloc d'instructions **doit** être faite **avant** la première instruction de cette procédure ou de ce bloc.

Note : on a parfois tendance à croire que le fait de déclarer une variable locale à un bloc d'instructions rend le programme moins gourmand en mémoire, l'espace nécessaire pour stocker la donnée n'étant utilisé que pendant l'exécution du bloc concerné. Ceci est malheureusement inexact ! En effet, **toutes** les variables locales sont créées en début de procédure ; ce n'est que dans le cas d'un compilateur optimisé, et si l'on a plusieurs variables locales de même type dans des blocs d'instructions différents que l'on peut gagner un peu de place mémoire.

3.1.5.2 Paramètres d'une procédure

Les paramètres d'une procédure (on parle souvent d'*arguments* de la procédure) se comportent comme des variables locales à la procédure : ils ne sont donc accessibles qu'au sein de la procédure elle-même (notons que la procédure peut tout-à-fait modifier la valeur de ces variables).

Leur particularité est de recevoir une valeur initiale **au moment de l'appel de la procédure** : ces valeurs peuvent donc être différentes lors de deux appels différents au sein du programme ; ceci est donc foncièrement différent des variables locales initialisées de la procédure, qui recoivent les mêmes valeurs à chaque appel de la procédure.

Enfin, insistons sur le fait qu'*il importe, lors d'un appel, de passer le nombre correct de paramètres, ainsi que des paramètres dont les types sont conformes aux types attendus par la procédure.*

3.1.5.3 Notion de prototype

La description d'une procédure incluant le type du résultat et les types des arguments est dite *prototype* de la procédure. C'est une *déclaration*, au même titre que « `int C;` ». En principe, toute procédure doit être déclarée, la déclaration se composant du prototype de la fonction suivi d'un point-virgule. La procédure PGCD se déclare ainsi :

```
int pgcd(int A, int B);
```

Il n'est en fait pas nécessaire de fournir le nom des paramètres ; la déclaration peut donc s'écrire :

```
int pgcd(int, int);
```

Dans la pratique, la déclaration, toujours utile, n'est indispensable que si la procédure est utilisée dans le texte du programme source avant que n'apparaisse sa définition. Pour cette raison, on place souvent dans le programme source la définition des procédures avant leur première utilisation.

On notera que le mot-clef `void` désigne un type de données particulier, de taille 0, qui indique qu'une fonction ne fournit pas de résultat, comme dans `void toto(int)`, ou n'a pas de paramètre, comme dans `int clock(void)`.

En résumé, un prototype ne décrit pas comment une procédure est programmée, mais simplement comment et dans quelles conditions elle peut être utilisée.

Enfin, on notera qu'un programme C comporte obligatoirement une procédure de nom `main`. C'est cette procédure (dont on ne peut pas choisir le nom) qui est appelée lorsque l'on demande l'exécution du programme. En principe, le prototype de cette procédure est le suivant :

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

Nous verrons ultérieurement ce que désigne exactement une déclaration telle que « `char * argv[]` ». Notons toutefois que ce prototype **impose** un résultat à cette procédure : c'est pour cette raison que l'on **doit** toujours terminer la procédure `main` par une instruction de type `return X;`

3.1.5.4 Compilation séparée

Les grosses applications développées en langage C peuvent atteindre quelques centaines de milliers de lignes de code, voire quelques millions de lignes. Ces applications sont développées par des équipes pouvant comporter des dizaines de programmeurs. On ne peut guère envisager de voir un programmeur travailler sur un programme comportant des millions de lignes de code, tandis que les autres attendent qu'il ait fini ses modifications pour corriger à leur tour les procédures dont ils sont responsables !

La solution consiste à répartir l'ensemble du programme source en plusieurs fichiers séparés, qui peuvent être édités et compilés séparément. Il est bien sûr nécessaire de rassembler ensuite les résultats de ces compilations pour pouvoir procéder à l'exécution du programme.

Imaginons que notre programme de calcul de *PGCD* soit jugé assez complexe pour relever d'une telle approche. Il serait possible de placer, par exemple, la procédure `pgcd` dans l'un de ces fichiers, et la procédure `main` dans un autre. Que faut-il faire pour que cette solution « fonctionne » ?

Tout d'abord, nous allons créer un fichier source contenant le programme « principal », celui dans lequel la procédure `main` est déclarée, et puisque cette procédure `main` va faire appel à la procédure `pgcd`, il faut qu'elle connaisse le *prototype* de cette fonction. Nous allons donc insérer, avant la définition de `main`, le *prototype* suivant :

```
extern int pgcd(int, int);
```

Le mot-clef `extern` indique que la procédure `pgcd` n'est pas définie dans ce fichier, mais dans un autre, tout en fournissant son prototype, qui permet au compilateur de vérifier que l'utilisation de cette procédure est conforme à sa définition.

Si le fichier contenant la procédure `main` a pour nom `prog3.c`, nous le compilerons par la commande :

```
[P]$ gcc -c prog3.c
```

Notons l'option « `-c` » qui indique que l'on désire seulement *compiler* le fichier `prog3.c`. Le résultat de l'opération est un nouveau fichier, de nom `prog3.o` qui contient le code objet de notre programme.

Il faut ensuite créer un second fichier source, contenant le texte de la procédure `pgcd`, et que nous nommerons par exemple `pgcd.c`. De la même manière que ci-dessus, nous compilerons ce fichier par :

```
[P]$ gcc -c pgcd.c
```

Là encore, un fichier, de nom `pgcd.o`, va être construit par le compilateur.

Nous pouvons alors créer le programme exécutable, par une commande telle que :

```
[P]$ gcc prog3.o pgcd.o -o prog3
```

Notons que cette fois-ci, ce sont les noms des fichiers contenant les codes objet (donc, se terminant par l'extension « `.o` »), que nous transmettons à la commande `gcc`. Le compilateur générera le fichier `prog3`, qui contient le programme directement exécutable.

Nous avons en fait décomposé la création du programme exécutable en deux étapes : la première consiste à compiler indépendamment les divers modules sources pour obtenir les modules objets correspondants ; la seconde rassemble ces modules objets (plus, éventuellement, des modules objets dits « de bibliothèque » contenant les procédures définies par le standard C) pour fournir le module exécutable. Cette seconde étape est souvent dit « édition des liens ».

Cette approche permet ainsi de modifier l'une ou l'autre des procédures, sans avoir besoin de recompiler systématiquement tout le code source de l'application.

3.1.5.5 Fichiers d'inclusion

Nous avons déjà présenté la commande « `include` » du préprocesseur. Cette commande a pour but d'aller chercher le fichier précisé en paramètre, et de l'insérer, dans le corps du programme, pour l'étape de compilation. La commande admet les deux syntaxes suivantes :

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

Dans la première forme, le fichier de nom « `stdio.h` » est recherché dans les répertoires contenant les déclarations des procédures prédéfinies du systèmes, ici, certaines opérations d'entrées-sorties. Dans la seconde forme, le fichier, ici « toto.h » est cherché d'abord dans le répertoire courant (puis dans les répertoires du système, comme ci-dessus) ; un tel fichier a en général été créé par l'utilisateur.

De tel fichiers, dits souvent *fichiers d'inclusion*, peuvent contenir n'importe quelles instructions du langage C. En général cependant, l'on n'y place que des déclarations et d'autres instructions destinées au préprocesseur.

Il nous serait possible, pour simplifier l'utilisation de la procédure `pgcd`, de définir son prototype dans un tel fichier. Créons le fichier « `pgcd.h` », contenant la seule ligne :

```
extern int pgcd(int, int);
```

Dans le fichier « `prog3.c` », celui qui contient le programme principal, nous pouvons dès lors remplacer la ligne définissant le prototype de la fonction par la ligne suivante :

```
#include "pgcd.h"
```

Notre programme principal devient :

```
#include <stdio.h>
#include "pgcd.h"
int main(int argc, char * argv[])
{
 int A, B;
 int R;
  A = 834389;
  B = 944353;
  R = pgcd(A,B);
  printf("La valeur du PGCD est %d\n", R);
  return 0;
}
```

Le gain d'espace est dans ce cas précis assez réduit. En général un module objet va contenir plusieurs procédures, et le fichier « include » correspondant va contenir l'ensemble des déclarations de ces procédures.

On notera que les fichiers d'inclusion utilisent par convention l'extension « `.h` ». Ceci simplifie la vie de l'utilisateur, qui sait que lorsqu'il a dans un répertoire les trois fichiers suivants :

```
pgcd.c   pgcd.h   pgcd.o
```

Ceux-ci représentent en principe un *module source*, le *fichier d'inclusion* contenant les prototypes des procédures du module source, et enfin le *module objet* correspondant.

3.1.6 Quelques procédures de C

Nous avons déjà utilisé, dans les cours antérieurs, quelques procédures du système, en particulier celles qui fournissent des services d'entrées sorties, `printf` et `puts`.

Nous proposons ici la description de quelques unes des procédures d'usage relativement fréquent dans les applications.

3.1.6.1 Opérations mathématiques

Pour les calculs scientifiques, C propose une vaste bibliothèque de procédures, dont certaines sont décrites à la figure 3.4. On notera que les paramètres et les résultats sont de type `double` (sur les machines que nous utilisons, ces nombres flottants sont représentés sur 8 octets, ce qui fournit 16 à 17 chiffres décimaux significatifs).

Plutôt que de fournir dans chaque programme l'ensemble de ces fonctions, l'environnement de programmation permet :

- d'inclure dans les programmes sources des fichiers de déclarations des prototypes ; nous l'avons fait avec le fichier `stdio.h`.
- d'inclure dans les programmes exécutables des bibliothèques contenant ces fonctions pré-compilées, au moyen du paramètre `-l`*nom*., par exemple `-lm` pour les opérations mathématiques.

L'utilisation de ces opérations mathématiques nécessite donc *l'inclusion de leur déclaration*, obtenue en insérant la ligne suivante en tête du programme :

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

Par ailleurs, il est nécessaire d'indiquer, lors de la phase d'édition des liens, que ces procédures sont à prendre dans la *bibliothèque mathématique standard* du système, en incluant l'option `-lm`.

Nom	**Prototype**	**Description**
`acos`	`double acos(double x)`	arc cosinus du paramètre
`asin`	`double asin(double x)`	arc sinus du paramètre
`atan`	`double atan(double x)`	arc tangente du paramètre
`atan2`	`double atan2(double x, double y)`	arc tangente de `x/y`
`ceil`	`double ceil(double x)`	plus grand entier $\leq$ au paramètre
`cos`	`double cos(double x)`	cosinus du paramètre
`cosh`	`double cosh(double x)`	cosinus hyperbolique du paramètre
`exp`	`double exp(double x)`	exponentielle du paramètre
`fabs`	`double fabs(double x)`	valeur absolue du paramètre
`floor`	`double floor(double x)`	plus petit entier $\geq$ au paramètre
`fmod`	`double fmod(double x, double y)`	reste de la division de x par y.
`log`	`double log(double x)`	logarithme du paramètre
`log10`	`double log10(double x)`	logarithme en base 10 du paramètre
`pow`	`double pow(double x, double y)`	x^y
`sin`	`double sin(double x)`	sinus du paramètre
`sinh`	`double sinh(double x)`	sinus hyperbolique du paramètre
`sqrt`	`double sqrt(double x)`	racine carrée du paramètre
`tan`	`double tan(double x)`	tangente du paramètre
`tanh`	`double tanh(double x)`	tangente hyperbolique du paramètre

FIG. 3.4 – Quelques procédures mathématiques du Langage C

Nom	Prototype	Description
abs	`int abs(int p)`	Valeur absolue du paramètre
atof	`double atof(char* s)`	Conversion caractères vers flottant
atoi	`int atoi(char* s)`	Conversion caractères vers entier
atol	`long atol(char* s)`	Conversion caractères vers entier
exit	`void exit(int cr)`	Arrêt du programme, fourniture d'un code de retour
labs	`long labs(long p)`	Valeur absolue du paramètre
rand	`int rand(void)`	Fournit un entier pseudo-aléatoire
srand	`void srand(int seed)`	Initialisation du générateur de nombres pseudo-aléatoires

FIG. 3.5 – Quelques utilitaires de C

Notes

- La fonction `fmod` fournit le reste de la division à quotient entier ; le diviseur doit être non nul. Ainsi, `fmod(4.1,1.5)` fournit comme résultat `1.1`. Le résultat est du signe du premier paramètre.
- la fonction `atan2` fournit l'arc dont le sinus est $k \times x$ et le cosinus $k \times y$, avec $k > 0$.
- Le résultat de certaines fonctions, lorsqu'il sort du domaine de définition des procédures, peut être représenté par des configurations spécifiques qui s'impriment sous la forme `inf` ("infini", par exemple le résultat d'une division par zéro) ou `nan` ("not a number").

3.1.6.2 Opérations utilitaires

D'autres procédures fournies par le système sont à vocation "utilitaire". La figure 3.5 en présente certaines. L'utilisation de ces opérations nécessite l'inclusion de leur déclaration, obtenue en insérant la ligne suivante en tête du programme :

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

Notes

- L'opération `rand` fournit les éléments successifs d'une séquence de nombres pseudo-aléatoires, calculés à partir d'un *germe* qui est 1 par défaut. La même séquence est donc fournie à chaque nouvelle exécution du programme.

– L'opération `srand` permet de modifier le *germe* (1 par défaut) utilisé par l'opération `rand`. On peut initialiser ce germe avec une valeur différente à chaque exécution d'un programme pour éviter d'obtenir, lors d'une simulation, des résultats strictement identiques à chaque exécution. Par exemple, la formule magique :

```
time((time_t *)0)
```

fournit le nombre de secondes écoulées depuis le 1er janvier 1970. Cette valeur peut être utilisée comme paramètre de `srand()`, en début d'exécution du programme, pour obtenir des suites aléatoires différentes à chaque exécution du programme (pour utiliser `time`, il faut inclure dans le programme les déclarations de `time.h`).

3.2 À retenir

– définition de procédure
type-du-résultat nom (*liste des paramètres*)
`{`
corps de la procédure
`}`
– paramètre
type-du-paramètre nom-du-paramètre
– type `void`
utilisé pour indiquer l'absence de paramètres ou de résultats
– sortie et fourniture de résultat :
  – `return ;` (fonction sans résultat)
  – `return` *expression* `;`
  (l'expression doit fournir une valeur du type déclaré pour la procédure)
– variables locales
elles sont déclarées au début du bloc constituant la procédure.
– appel de procédure
  – procédures sans paramètre
  nom`( )`
  – procédure avec paramètres
  nom`(`*liste de valeurs*`)`
Un appel de procédure doit respecter le nombre, l'ordre, et les types des paramètres.
– déclaration de procédure : au moyen du *prototype*
type-du-résultat nom `(`*liste des paramètres*`) ;`
Dans le prototype, les *noms* de paramètres sont facultatifs.
– déclaration de procédure externe : au moyen du mot-clef `extern`
– inclusion de déclarations : commande du préprocesseur

```
#include "file.h"
```

– procédures prédéfinies du langages : la déclaration de leurs prototypes se trouve dans les fichiers `math.h` (procédures mathématiques) et `stdlib.h` (autres procédures «standard»)

3.3 Travaux pratiques

Chaque étudiant enverra les programmes réalisés au cours de ces travaux pratiques sous la forme d'un *mél* unique, comportant les textes des programmes, séparés par trois lignes blanches les uns des autres, à

l'adresse `girardot@emse.fr`. Le sujet du *mél* comportera le **nom de l'étudiant**, et la mention TP 3.

3.3.1 Exercice 1

Écrire une fonction qui calcule le *PPCM* de deux nombres entiers. Écrire une fonction qui indique si deux nombres sont premiers entre eux.

3.3.2 Exercice 2

Reprendre la procédure PGCD définie au paragraphe 3.3, page 49. Que se passe-t-il si l'on effectue l'appel :

```
pgcd(0,11)
```

Comment protéger la procédure contre tout paramètre incorrect ? Comment une procédure peut-elle signaler une erreur qu'elle détecte au programme appelant ?

3.3.3 Exercice 3

Écrire (en vous basant sur un travail antérieur) une procédure `premier(n)` qui indique si son paramètre est un nombre premier. Réaliser le programme qui affiche les N premiers nombres premiers (avec N au moins égal à 100).

3.3.4 Exercice 4

Identifier, dans le projet, les parties qui vont se transformer en une (ou plusieurs) procédures. Réfléchir en particulier aux paramètres qu'il conviendra de transmettre à ces procédures.

Chapitre 4

Tableaux

4.1 Introduction

Ce cours va nous permettre d'aborder les *tableaux*, qui sont des collections d'objets de même nature, les éléments d'un tableau de taille N étant désignés par les entiers de l'intervalle $[0, N-1]$.

4.1.1 Un premier exemple

Imaginons que nous souhaitions traiter les notes (numériques) d'un groupe de 10 élèves : calculer la meilleure note, la moins bonne, la moyenne, la médiane, convertir en notation-lettre...

Pour stocker ces notes, il est possible d'utiliser autant de variables que de notes, mais ceci présente de nombreux inconvénients :

- la déclaration de ces variables devient très vite fastidieuse !
- le traitement est également très lourd puisque l'on ne dispose d'aucun moyen (à part copier-coller le code, ce qui est en plus source d'erreurs) de traiter chaque note ; on ne dispose en particulier d'aucun élément du langage pour traiter globalement ces notes ;
- si le nombre de notes vient à varier, l'écriture des codes de traitement est encore alourdie.

C'est pour ces raisons que les langages de programmation de haut niveau (dont C fait bien sûr partie), offrent tous la possibilité de représenter des collections de données de même type : les tableaux.

Supposons donc que nous ayons à traiter les 10 valeurs :

```
12 8 13 10 6 8 19 20 9 16
```

Nous allons représenter ainsi ces données par une variable de type tableau d'entiers ; en C, une telle variable se déclare comme suit :

```
int notes[10];
```

Par rapport à une déclaration de variable scalaire, nous avons simplement ajouté, derrière le nom de la variable, un nombre entre crochets. Ce nombre, qui est obligatoirement un entier, indique le nombre d'éléments réservés pour le tableau.

Un élément du tableau est alors désigné par l'expression `notes[`*indice*`]`, dans laquelle *indice* est une expression quelconque dont la valeur doit être entière et comprise entre 0 et 9. Attention : ne pas oublier que si le tableau est déclaré de taille N, le dernier élément est celui d'indice $N-1$ (ceci est fréquemment source d'erreurs).

L'initialisation des éléments du tableau peut s'écrire :

```
notes[0]=12; notes[1]=8; notes[2]=13; ... notes[9]=16;
```

Comme pour toute variable, on peut, au moment de la déclaration, initialiser les valeurs du tableau : une telle initialisation s'effectue par :

```
int notes[10] = {12, 8, 23, 24, 6, 8, 19, 20, 9, 4};
```

Le fragment de programme suivant permet de calculer, par une boucle, la moyenne des notes du tableau (les variables `somme`, `moyenne` et `indice` ayant été déclarées en `int`) :

```
somme = 0;
indice = 0;
while (indice < 10)
{
  somme = somme+notes[indice];
  indice = indice+1;
}
moyenne = somme/10;
```

Ou encore :

```
somme = 0;
for (indice=0; indice<10; indice=indice+1)
  somme = somme+notes[indice];
moyenne = somme/10;
```

On notera la syntaxe de cette dernière forme :

`for (`*valeur_initiale* `;` *test_de_répétition* `;` *gestion_de_l'indice*`)` *instruction*

qui est équivalente à ...

valeur_initiale `; while (`*test_de_répétition*`) {` *instruction gestion_de_l'indice* `; }`

4.1.2 Les tableaux

4.1.2.1 Caractéristique de base

Les tableaux sont représentés par des arrangements contigus de cellules de mémoire. L'encombrement d'un tableau est le produit du nombre d'éléments par la taille d'un élément. Un tableau de 10 `int` utilise donc 10×4, soit 40 octets de la mémoire sur nos machines.

La norme du langage précise que les éléments du tableau sont rangés en mémoire par valeurs croissantes de leurs indices.

4.1.2.2 Déclaration de tableau

Lors d'une déclaration d'un tableau dans un programme, la taille du tableau doit être indiquée d'une manière ou d'une autre :

```
int notes[10];
int titi[10] = {2,3,5,6,7,8,10,13,14,19};
int tutu[] = {1,2,3,4,5};
```

On notera dans ce dernier cas que la taille du tableau n'est pas indiquée, mais que le nombre d'éléments (5) peut être déduit par le compilateur à partir des valeurs indiquées. Les déclarations suivantes, par contre, sont incorrectes ou provoqueront des erreurs à l'exécution :

```
int jojo[];
int juju[5]={1,2,3,8,10,12,15};
```

La première n'indique pas la taille du tableau ; la seconde tente d'initialiser avec 7 valeurs un tableau déclaré à 5 éléments.

Enfin, l'exemple suivant est correct :

```
int premiers[1000]={2,3,5,7};
```

Dans le tableau ainsi déclaré, de 1000 éléments, seuls les 4 premiers reçoivent une valeur. Il n'est pas possible de présumer de la valeur des éléments non initialisés : certains compilateurs peuvent décider d'affecter une valeur par défaut (0, pour des entiers), mais un programme ne doit en aucun cas considérer ceci comme un fait acquis.

Notons enfin que la taille du tableau doit être connue *au moment de la compilation*. Il est ainsi incorrect d'utiliser une variable pour déclarer la taille d'un tableau, et le code suivant provoquera une erreur :

```
int n;
int tab[n];
```

4.1.2.3 Accès aux éléments des tableaux

Dans une expression, la notation `tab[i]` permet de référencer l'élément `i` du tableau `tab`. Cette expression s'apparente à une référence à un nom de variable, et peut donc être utilisée dans la partie gauche d'une affectation (ce que l'on appelle une *left-value* ou *lvalue*) :

```
tab[i] = expression;
```

cette écriture ayant pour effet de remplacer la valeur de l'élément `i` de `tab` par celle de l'expression.

Notons aussi que des écritures telles que `tab[i]++` sont légales et ont pour effet d'incrémenter l'élément `i` du tableau `tab`.

C'est naturellement une erreur que d'utiliser comme indice une valeur non entière, négative, ou encore supérieure ou égale à la taille du tableau.

Attention : cette erreur ne sera pas signalée par le compilateur (ce n'est pas une erreur de syntaxe, mais une erreur « sémantique »), mais peut entraîner une erreur d'exécution, puisque le programme accède ainsi (en lecture ou en écriture), à un emplacement qu'il n'est pas supposé consulter ou modifier.

4.1.2.4 Tableaux et adresses

On a vu que l'on peut accéder « individuellement » aux éléments d'un tableau. Il n'est par contre pas possible de manipuler le tableau « globalement » ; en particulier, il n'est pas possible d'affecter une valeur à un tableau, et les écritures ci-dessous provoqueront des erreurs de compilation :

```
int tab1[10],tab2[10];
tab1={0,1,2,3,4,5,6,7,8,9}; /* ERREUR !! */
tab1=tab2; /* ERREUR !! */
```

En fait, lorsque l'on utilise le nom d'une variable de type tableau sans faire référence à un indice de ce tableau, cette expression désigne en fait *l'adresse mémoire du premier élément*. Ainsi, le code suivant est parfaitement correct :

```
int tab[10],*adr,i;
adr=tab; /* OK */
i=*adr; /* equivaut a i=tab[0] */
```

la première instruction consistant à stocker dans la variable `adr` de type adresse d'entier l'adresse mémoire du premier élément du tableau `tab`, qui est bien de type entier.

Il est par ailleurs totalement licite d'utiliser la notation indicielle (valeur entière entre crochets) avec une variable définie non pas comme un tableau, mais comme une adresse. Ainsi, le code suivant est tout à fait correct :

```
int tab[10],*adr;
adr=tab;
if (adr[2]==4) /* si le deuxieme element du tableau vaut 4 */
  ...
```

Par contre, l'écriture suivante est incorrecte :

```
int tab[10],*adr;
tab=adr; /* ERREUR !! */
```

car ceci reviendrait à vouloir changer l'adresse en mémoire où est stocké le tableau `tab`, ce qui n'est pas possible (c'est le compilateur qui gère cette adresse). On a ainsi tradition de dire que « le nom d'un tableau est une adresse constante ».

On verra par la suite que l'on mélange allègrement ces deux notions, utilisant selon les préférences du programmeur la version tableau ou la version adresse. Il ne faut cependant pas oublier que ces deux notions ne sont pas tout à fait identiques.

4.1.2.5 Tableaux de caractères

Nous avons en fait déjà manipulé (sans le savoir) un type de tableau fort utile, les tableaux de caractères. Ceux-ci permettent la définition de chaînes de caractères, utilisées pour la représentations des données alphanumériques, des messages, etc.

Un tableau de caractères peut se déclarer sous des formes telles que :

```
char tab1[10];
char tab2[10]={0,1,2,3,4,5,6,7,8,9};
char tab3[10]={'H','e','l','l',0x6f,48,49,50,51,52};
char tab4[10]="ABCD";
```

Les tableaux `tab2` et `tab3` sont initialisés avec des valeurs entières (qui doivent appartenir à l'intervalle $[-128, 127]$), fournies ici sous forme décimale (`50`), hexadécimale (`0x6f`) ou caractère (`'e'`), représentant en l'occurrence les codes des caractères '2', 'o' et 'e' respectivement.

Les éléments du tableau `tab4` sont initialisés avec les valeurs représentées par la chaîne de caractères `"ABCD"`. **Attention** ! Cette chaîne

contient 5 valeurs, qui sont les quatre lettres 'A', 'B', 'C' et 'D', et le nombre 0 que le compilateur utilise comme *marqueur* de fin de chaîne. Nous reviendrons sur cette convention dans la section 4.1.3.

4.1.2.6 Tableaux et procédures

Un tableau peut être utilisé comme paramètre d'une procédure :

```
int proc(int titi[10])
{ ... }
...
int toto[10];
proc(toto);
```

Dans cet exemple, le paramètre `titi` de la procédure `proc` est déclaré en tant que tableau de 10 éléments. La procédure est ensuite appelée avec comme paramètre un tableau de taille adéquate.

Il a été indiqué que toute utilisation du seul nom du tableau fait référence à l'adresse mémoire du premier élément : c'est donc l'adresse mémoire du tableau qui est passée à la procédure. Ceci a deux conséquences immédiates :

- dans la compilation de la procédure `proc`, le compilateur ne tient pas compte de la taille déclarée pour le paramètre (car il n'a pas besoin d'allouer la place mémoire pour le tableau, seule l'adresse est transmise et stockée dans la variable locale `titi`), et il est équivalent d'écrire :

  ```
  int proc(int titi[])
  ```

  Attention : cette notation n'est possible que pour les arguments d'une procédure, pas pour déclarer des variables locales ! On trouvera également souvent la notation équivalente :

  ```
  int proc(int * titi)
  ```

  nous avons déjà signalé la grande similitude entre tableau et adresse.
- puisque la procédure accède à l'adresse mémoire du tableau, toute modification du contenu du tableau dans la procédure modifie effectivement le tableau passé en paramètre ;

Enfin, signalons qu'une procédure à laquelle on a passé un tableau en argument n'a *aucun* moyen (par le langage lui-même) de connaître la taille du tableau : il n'y a pas d'opérateur *taille de tableau*, et l'opérateur `sizeof()` déjà mentionné appliqué à un tableau argument de procédure rendra comme résultat la taille d'une adresse mémoire, soit 4 octets dans nos implémentations. Ceci impliquera donc toujours que :

– soit la procédure connaît *a priori* la taille du tableau ;
– soit un autre argument de la procédure en précise la taille.

4.1.2.7 Exemples

Voici un exemple de procédure calculant la somme des éléments d'un tableau reçu comme paramètre. Comme indiqué ci-dessus, le deuxième argument précise la taille du tableau.

```
int pruc(int titi[], int n)
{
  int i;
  int somme=0;
  for (i=0; i<n; i++)
   somme = somme+titi[i];
  return somme;
}
```

Retour à la procédure `pruc` Les deux paramètres permettent de transmettre à la procédure le tableau (en fait, son adresse), ainsi que le nombre d'éléments de ce tableau. Des appels valides de la procédure sont par exemple :

```
pruc(toto,10);
printf("%d\n", pruc(toto,10));
```

Dans le premier cas, le résultat fourni par la fonction n'est pas utilisé. Dans le second cas, il est imprimé par la procédure `printf`.

On notera que l'appel :

```
pruc(toto,8);
```

permet de calculer la somme des 8 premiers éléments du tableau, bien que `toto` ait été déclaré de dimension 10.

Note Une écriture équivalente à la première est :

```
pruc(&toto[0],10);
```

Le premier paramètre se lit "adresse de l'élément zéro du tableau `toto`". Cette notation permet de transmettre l'adresse d'éléments autres que le premier du tableau. Ainsi, puisque les éléments sont rangés en mémoire par valeurs croissantes des indices, l'expression :

```
pruc(&toto[2],8);
```

permet de calculer la somme des éléments 2 à 9 du tableau.

Voici un second exemple : la procédure `minmax` calcule le minimum et le maximum des éléments d'un tableau passé en paramètre. Le résultat est placé dans un autre tableau, également passé en paramètre :

```
int minmax(int tab[], int res[], int n)
{
  int i;
  int max, min;
  max = min = tab[0];
  for (i=1; i<n; i++)
  {
    if (min > tab[i])
      min = tab[i];
    if (max < tab[i])
      max = tab[i];
  }
  res[0]=min;
  res[1]=max;
}
```

Voici un programme utilisant cette procédure

```
#include <stdio.h>
int main(int argc, char * argv[])
{
 int juju[10]={1,2,3,5,6,8,10,12};
 int res[2];
  minmax(juju, res, 10);
  printf("%d %d\n", res[0], res[1]);
  return 0;
}
```

4.1.3 Chaînes de caractères

4.1.3.1 Introduction

Une chaîne de caractères est une suite d'octets non nuls, se terminant pas un octet égal à 0, dit souvent *null*. En C, une telle chaîne est représentée par un tableau de `char` : *par exception à la règle générale, il est possible de connaître la taille de la chaîne, il suffit de compter les caractères jusqu'au caractère 0 qui marque la fin*.

Le compilateur reconnaît pour les chaînes de caractères une notation de constante débutant et se terminant par une double quote, encadrant la chaîne à définir :

```
"En voici une"
```

Lorsque cette convention est utilisée, le caractère *double-quote* " lui-même doit être précédé d'un caractère `\`, dit *caractère d'échappement*, ou *back-slash*, ce dernier caractère, pour être représenté, devant lui-même être précédé d'un autre back-slash. Le caractère d'échappement permet la représentation de certains caractères dits "*de contrôle*" : ainsi, le *passage à la ligne* se note `\n`.

Dans un programme C, une chaîne de caractères représente une référence à un tableau. Une notation telle que `"ABCDE"[3]` représente l'élément d'indice 3 du tableau, c'est à dire le caractère `'D'` dont la représentation décimale est 68.

Notons encore que les compilateurs C effectuent une concaténation implicite de chaînes de caractères qui se suivent dans le texte d'un programme C. Ainsi, les deux lignes suivantes définissent-elles la même chaîne :

```
"ABC" "XY" "012345"
"ABCXY012345"
```

Ceci permet de disposer agréablement les longues chaînes sur plusieurs lignes, afin d'augmenter la lisibilité des programmes.

Enfin, rappelons que la plus grande attention doit être portée à la manipulation de chaînes contenant des caractères en dehors du code ASCII : caractères accentués, signes monétaires...Les résultats peuvent dépendre de l'environnement : système d'exploitation, compilateur, variables d'environnement.

4.1.3.2 Opérations sur chaînes de caractères

La figure 4.1 décrit certaines des opérations de base (procédures standards) relatives aux chaînes de caractères. Les prototypes de ces fonctions sont décrits dans le fichier d'inclusion `string.h`.

Notes

1. ces opérations reposent sur la convention implicite qu'une chaîne de caractères est représentée par une suite (éventuellement vide) d'octets de valeur non nulle, suivie par un octet de valeur égale à 0. Si l'un des paramètres ne respecte pas ce schéma, le comportement est indéterminé (mais on peut prédire des problèmes...)
2. dans les opérations entraînant le transfert d'éléments (telles `strcat`, `strcpy`, etc.) la destination est le premier paramètre de la procédure.

Nom	Prototype	Description
`strcat`	`char * strcat(char * d, const char * s)`	Concaténation de deux chaînes
`strcmp`	`int strcmp(const char * d, const char * s)`	Comparaison de deux chaînes
`strcpy`	`char * strcpy(char * d, const char * s)`	Copie d'une chaîne
`strlen`	`int strlen(const char * s)`	Longueur d'une chaîne
`strncat`	`char * strncat(char * d, const char * s, int n)`	Concaténation de deux chaînes
`strncmp`	`int strncmp(const char * d, const char * s, int n)`	Comparaison de deux chaînes
`strncpy`	`char * strncpy(char * d, const char * s, int n)`	Copie d'une chaîne

FIG. 4.1 – Quelques opérations sur chaînes de caractères

3. le mot-clef `const` précise que le paramètre transmis désigne une zone de mémoire qui *ne sera pas modifiée* par la procédure.
4. les versions avec `n` (`strncat`, `strncpy`, `strncmp`) limitent l'opération à au plus `n` caractères ; attention : dans le cas de `strncpy`, si la chaîne source contient plus de `n` caractères, la chaîne résultat ne contient pas le `0` final !
5. les opérations *ne tiennent pas compte des tailles effectives des tableaux utilisés*. C'est au programmeur de s'assurer que la taille de l'opérande destination est suffisante pour recevoir l'ensemble des octets de la source.

4.1.3.3 Exemples de manipulation de chaînes de caractères

Déclaration d'une chaîne

```
char c1 [] = "Une suite d'octets";
```

Le compilateur déduit ici que la taille du tableau est de 19 octets (incluant l'octet final de valeur 0).

On aurait pu déclarer de façon équivalente :

```
char *c1 = "Une suite d'octets";
```

Longueur d'une chaîne Par convention, la longueur d'une chaîne de caractères est le nombre d'octets non nuls de la chaîne ; ainsi, la valeur de l'expression :

```
strlen("ABCD");
```

est 4 (le nombre d'octets non nuls), et non 5, nombre d'octets servant à représenter la chaîne.

Copie d'une chaîne

```
char c2[10];
...
strcpy(c2,"ABCD");
```

Dans ce fragment de programme, le tableau `c2`, de dix éléments, reçoit une copie des éléments de la chaîne `"ABCD"`. Après l'exécution, les éléments du tableau `c2` ont pour valeurs :

65	66	67	68	0	?	?	?	?	?

(Les points d'interrogation représentent les éléments *non modifiés* du tableau `c2`.) Ce serait une erreur d'exécuter :

```
strcpy(c2,"ABCDEFGHIJKLMNOP");
```

car le tableau `c2` n'est pas de taille suffisante pour recevoir une chaîne de 16 caractères, c'est-à-dire 17 valeurs.

On peut, pour se protéger de tels débordements, utiliser la procédure `strncpy` :

```
strncpy(c2,"ABCD",10);
```

qui permet d'indiquer que 10 octets au plus doivent être copiés (en comptant l'octet *null* de fin de chaîne). Attention, si la seconde chaîne est de taille supérieure ou égale au nombre maximum d'octets à copier, il n'y a pas insertion de *null* en fin de chaîne, ce qui risque de provoquer des erreurs ultérieures...

Concaténation L'opération de concaténation permet d'ajouter, au bout d'une chaîne de caractères, une copie d'une autre chaîne. L'opération modifie son premier opérande. Après l'exécution de :

```
strcat(c2,"XYZ");
```

les éléments du tableau `c2` ont maintenant pour valeurs :

65	66	67	68	88	89	90	0	?	?

Impression La spécification de format `%s` permet l'impression d'une chaîne :

```
printf("c2 vaut : \"%s\"\n", c2);
```

imprimera :

```
c2 vaut : "ABCDXYZ"
```

Comparaison L'opération de comparaison s'applique à deux chaînes de caractères. Le résultat est un entier qui est *négatif* si la première chaîne est *inférieure* à la seconde, *nul* si les deux chaînes sont *égales*, et *positif* si la première est *supérieure* à la seconde.

```
strcmp("ABCD","ABD")
```

Cette expression rend une valeur négative, car la chaîne `"ABCD"` est située avant la chaîne `"ABD"` dans l'ordre lexicographique (très précisément, `'C'` est situé avant `'D'` dans le code ASCII).

Une écriture possible en C de la procédure *strcmp* est la suivante :

```
int strcmp(char s1[], char s2[])
{
  int c1, c2;
  int i;
  for (i=0;;i++)
  {
    c1=s1[i]; c2=s2[i];
    if (c1 == 0 || c2 == 0 || c1!= c2)
      return c1-c2;
  }
}
```

4.1.4 Retour sur la procédure `main`

Nous avions indiqué que tout programme `C` devait comporter une procédure `main`, et nous pouvons désormais préciser quels en sont les arguments.

Rappelons que le prototype de cette procédure est :

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

Les paramètres (que tout le monde a coutume de dénommer `argc` et `argv` bien que, comme pour toute procédure, le nom puisse être choisi tout à fait arbitrairement) sont donc :

– un entier , qui est le nombre d'arguments de la ligne de commande au moment de l'exécution du programme (`argc` pour *argument count*) ;
– un tableau de chaînes de caractères, qui sont les arguments (`argv` pour *argument values*).

Ainsi, si le programme `prog` est lancé par la commande (dans un interpréteur de commandes de type shell ou tout autre moyen d'interfaçage)

```
prog arg1 arg2
```

la valeur de la variable `argc` sera de 3, et le tableau `argv` contiendra :

```
argv[0] = "prog"
argv[1] = "arg1"
argv[2] = "arg2"
```

NB1 : noter que le nom du programme lui-même fait partie des arguments, et que la valeur `argc` vaut donc au minimum 1 !

NB2 : tous les arguments sont du type chaîne de caractères ; si l'on souhaite utiliser un argument de type entier ou flottant, il faudra utiliser une fonction de conversion à l'intérieur du code source du programme ;

4.1.4.1 Utilisation des arguments de la fonction `main`

Il est bien sûr possible d'utiliser ces arguments. Ils permettent notamment de fournir au moment de l'exécution (et sans aucune entrée-sortie ou dialogue avec l'utilisateur) des informations au programme. D'ailleurs, chaque fois que nous utilisons un outil (éditeur de texte, compilateur, débogueur)

4.2 À retenir

– déclaration de tableau :
type nom[*dimension*] ;
– déclaration avec initialisation :
type nom[*dimension*]= {*liste de valeurs*} ;
– indice d'un élément de tableau
$0 \leq valeur < dimension$
– élément de tableau
nom[*indice*]
nom[*indice*]=*valeur*
– déclaration de pointeur
type * *nom* ;
– chaîne de caractères
notation spécifique pour un tableau de caractères
caractère «null» ajouté à la fin de la chaîne
`"ABCD"` équivaut au tableau contenant les cinq caractères de codes 65, 66, 67, 68 et 0.
– opérations sur chaînes :
définies dans `string.h`
– tableau et pointeurs
– tableau et procédure
passage des tableaux par adresse

4.3 Travaux pratiques

Chaque étudiant enverra les programmes réalisés au cours de ces travaux pratiques sous la forme d'un *mél* unique, comportant les textes des programmes, séparés par trois lignes blanches les uns des autres, à l'adresse `girardot@emse.fr`. Le sujet du *mél* comportera le **nom de l'étudiant**, et la mention TP 4.

4.3.1 Exercice 1

On se propose de déterminer la moyenne olympique d'une suite de N notes. Celles-ci sont données sous la forme d'un tableau. Il convient de supprimer de ces notes la plus élevée (ou l'une des plus élevées en cas de *ex-aequo*) et la plus faible, et de calculer la moyenne des $N-2$ restantes.

Écrire une fonction dont le prototype est le suivant :

```
double moy_olymp(int notes[], int N);
```

4.3.2 Exercice 2

Écrire une fonction qui réalise une permutation aléatoire d'un tableau d'entiers (on peut utiliser la fonction `rand`). Si possible, cette fonction ne doit pas créer de copie du tableau, mais le permuter « en place ».

Le prototype de cette fonction doit être :

```
void permute_tableau (int tab[],int N);
```

4.3.3 Exercice 3

On se propose de construire un histogramme décrivant la répartition des valeurs des éléments d'un vecteur tel que celui-ci :

```
{ 2, 3, 7, 6, 9,11,12,15,18,17, 14,13,12, 7,
8, 7, 5, 3, 2, 1, 1, 0, 2, 8,11,13,12,11, 6, 3};
```

Le programme affichera l'histogramme dans la fenêtre du terminal, en utilisant des blancs et des étoiles, comme dans l'exemple de la figure 4.2.

Écrire une fonction dont le prototype est :

```
void histog(int v[], int nb)
```

Modifier le programme afin de pouvoir choisir la hauteur de l'histogramme. Le prototype devient :

```
void histog(int v[], int nb, int H)
```

4.3.4 Exercice 4

On veut maintenant analyser le générateur de nombres aléatoires fourni par le système, `rand()`. On tirera 10000 valeurs aléatoires, comprises entre 0 et 50, dont on étudiera la répartition au moyen du générateur d'histogrammes de l'exercice 2.

4.3.5 Exercice 5

Modifier le programme précédent, afin de générer cette fois une *distribution gaussienne* qui sera calculée par le programme (on peut approcher une distribution gaussienne par une somme de douze valeurs aléatoires uniformes de [-0.5,0.5]). On tracera l'histogramme de la répartition.

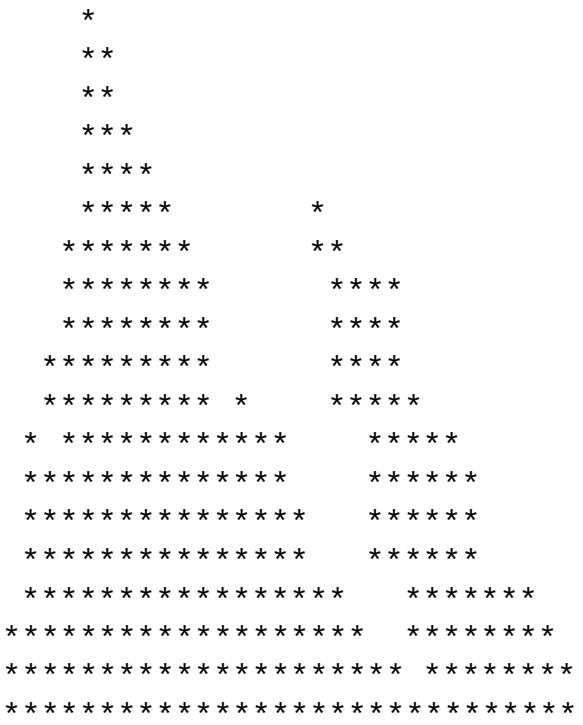

FIG. 4.2 – Histogramme

Chapitre 5

Du problème au programme

5.1 Cours

Les problèmes traités au chapitre 2, de nature mathématique (calcul d'un PGCD ou d'un sinus), étaient suffisamment simples pour être traités « en une seule passe », c'est-à-dire par l'écriture directe (après toutefois un peu de réflexion) du programme correspondant.

Il faut bien prendre conscience que dans la réalité, aucun problème n'est suffisamment simple pour être abordé ainsi. Il est très souvent nécessaire de procéder à une analyse permettant de décomposer le problème en sous-problèmes, puis éventuellement encore en sous-problèmes, jusqu'à aboutir à un problème pouvant être traité directement.

On retrouve ici une démarche très usuelle dans le domaine de l'ingénieur : que ce soit un système industriel destiné à traiter des matériaux pour élaborer des produits manufacturés, ou un programme complexe destiné à traiter des données pour construire d'autres données, la démarche générale d'analyse, de décomposition en sous-ensembles fonctionnels indépendants, est identique.

5.1.1 Calcul d'intervalle de temps

On cherche à calculer l'intervalle de temps (en jours, heures, minutes, secondes) séparant deux instants donnés. Chaque instant est défini par la donnée des jour, mois, année, heure, minute, seconde.

5.1.1.1 Une première décomposition

Déterminer directement ce nombre de jours, heures, minutes et secondes est assez difficile. On va donc décomposer ce travail en étapes plus simples.

On va dans un premier temps calculer le nombre de secondes entre les deux instants donnés : une fois ce nombre connu, de simples divisions et calculs de restes permettent d'obtenir les quatre données voulues.

Maintenant, pour calculer ce nombre de secondes, on va en fait calculer, pour chaque instant, l'intervalle (en nombre de secondes toujours) par rapport à une date de référence T_0. On utilise donc la relation :

$$T_2 - T_1 = (T_2 - T_0) - (T_1 - T_0)$$

On choisit comme date de référence le 1er janvier 1970 à 0h00. On sait en effet que c'est sous cette forme que sont stockées de nombreuses informations de date dans le système Unix, et on dispose en plus d'une fonction de bibliothèque `time` qui permettra de tester notre propre version de cette fonction.

On supposera donc dorénavant que les instants T_1 et T_2 sont postérieurs à cette date. On peut ainsi déja écrire une partie du code :

```
int delta_secondes (
 int J1, int M1, int A1, int h1, int m1, int s1,
 int J2, int M2, int A2, int h2, int m2, int s2)
{
  return N(J1,M1,A1,h1,m1,s1)-N(J2,M2,A2,h2,m2,s2);
}
```

On ne tiendra pas compte (pour l'instant) des problèmes de passage entre heure d'été et heure d'hiver : toutes les dates sont supposées être exprimées en temps universel.

5.1.1.2 Analyse de la fonction N

On notera pour la suite J, M, A, h, m et s les jour (de 1 à 31), mois (de 1 à 12), année (supérieure à 1970), heure (de 0 à 23), minute (de 0 à 59), seconde (de 0 à 59)[1]. On peut encore décomposer le problème en remarquant que la quantité recherchée est la somme :

- du nombre de secondes entre le 1er janvier 1970 à 0h00 et le 1er janvier de l'année A à 0h00 (ce nombre ne dépend que de A, on le note $N_1(A)$) ;
- du nombre de secondes entre le 1er janvier de l'année A à 0h00 et le 1er jour du mois M de l'année A à 0h00 (ce nombre ne dépend que de A et M, on le note $N_2(M, A)$) ;

[1] dans un souci de simplicité, on ne tiendra pas compte des secondes supplémentaires parfois ajoutées pour corriger les défauts de durée de révolution terrestre.

- du nombre de secondes entre le 1er jour du mois M de l'année A à 0h00 et le jour J du mois M de l'année A à 0h00 (ce nombre ne dépend que de J, on le note $N_3(J)$) ;
- du nombre de secondes entre le jour J du mois M de l'année A à 0h00 et le jour J du mois M de l'année A à l'heure h :m.s (ce nombre ne dépend que de H, M et S, on le note $N_4(h, m, s)$).

On peut ainsi écrire :

$$N(J, M, A, h, m, s) = N_1(A) + N_2(M, A) + N_3(J) + N_4(h, m, s)$$

On peut de plus remarquer que N_1, N_2 et N_3 sont des nombres de secondes correspondant à des jours « entiers » : il est ainsi possible d'écrire $N_i = Nj_i * 86400$, les Nj_i correspondant aux nombres de jours (et 86400 est le nombre de secondes par jour). On peut donc écrire la fonction N :

```
int N (int J,int M,int A,int h,int m,int s)
{
  return N_4(h,m,s)+86400*(Nj_1(A)+Nj_2(M,A)+Nj_3(J));
}
```

5.1.1.3 Calcul de $N_4(h, m, s)$

Ce calcul est très simple : on peut écrire directement la fonction :

```
int N_4 (int h,int m,int s)
{
  return s+60*(m+60*h);
}
```

5.1.1.4 Calcul de $Nj_3(j)$

Pas de difficulté particulière non plus ! On écrit :

```
int Nj_3 (int j)
{
  return j-1;
}
```

5.1.1.5 Calcul de Nj_1

On peut écrire :

$$Nj_1(A) = \sum_{a=1970}^{A-1} n_j(a)$$

où $n_j(a)$ désigne le nombre de jours de l'année a. On sait que les années comportent 365 jours, sauf les années bissextiles qui en comportent 366. On rappelle que, dans le calendrier grégorien, sont bissextiles les années divisibles par 4, sauf celles divisibles par 100, à moins qu'elles ne soient aussi divisibles par 400 [2].

On peut donc écrire une fonction que l'on note *bissextile(A)* permettant de tester si son argument est bissextile. Cette fonction permet ensuite d'écrire l'algorithme de calcul de Nj_1, écrit ci-dessous en `C` :

```
int n_j (int a)
{
  if (bissextile(a))
    return 366;
  return 365;
}

int Nj_1 (int annee)
{
int nb,a;
  for (nb=0,a=1970;a<annee;a++)
    nb+=n_j(a);
  return nb;
}
```

Le code de la fonction *bissextile* peut lui-même s'écrire :

```
int bissextile (int annee)
{
  if (annee % 4)
    return 0;
  if (annee % 100)
    return 1;
  if (annee % 400)
    return 0;
  return 1;
}
```

5.1.1.6 Calcul de Nj_2

Calcul également non trivial. En effet, les mois ont un nombre de jours variable, et il n'existe pas vraiment de « formule » permettant de calculer ce nombre de jours en fonction du numéro du mois. Il est alors

[2] Ceux intéressés par les problèmes d'ajustement de calendrier peuvent consulter l'URL `http ://www.emse.fr/~roelens/calendar.txt`

plus pratique d'écrire une fonction qui effectue un calcul spécifique en fonction de chaque mois.

Notons au passage que le calcul est modifié à partir du mois de mars si on est dans une année bissextile, d'où l'utilité de passer en paramètre l'année ! Voici une (ce n'est bien sûr pas la seule) façon d'écrire cette fonction :

```
int Nj_2 (int mois,int annee)
{
int nb=0;

  switch (mois) {
    case 12:
      /* on ajoute le nb de jours de novembre */
      nb+=30;
    case 11:
      /* on ajoute le nb de jours d'octobre */
      nb+=31;
    case 10:
      /* on ajoute le nb de jours de septembre */
      nb+=30;
    case 9:
      /* on ajoute le nb de jours d'aout */
      nb+=31;
    case 8:
      /* on ajoute le nb de jours de juillet */
      nb+=31;
    case 7:
      /* on ajoute le nb de jours de juin */
      nb+=30;
    case 6:
      /* on ajoute le nb de jours de mai */
      nb+=31;
    case 5:
      /* on ajoute le nb de jours d'avril */
      nb+=30;
    case 4:
      /* on ajoute le nb de jours de mars */
      nb+=31;
    case 3:
      /* on ajoute le nb de jours de fevrier */
      /* 29 si annee bissextile, 28 sinon */
      if (bissextile(annee))
```

```
        nb+=29;
      else
        nb+=28;
    case 2:
      /* on ajoute le nb de jours de janvier */
      nb+=31;
    case 1:
      return nb;
  }
}
```

Remarque 1 : on réutilise ici la fonction `bissextile`, déjà utilisée pour la fonction Nj_1. Cette réutilisation est l'un des avantages du découpage en fonctions : une fonction représente une suite d'instructions utilisable autant de fois que souhaité.

Remarque 2 : cette fonction dépend *stricto sensu* non pas de l'année, mais de la bissextilité de l'année.

Remarque 3 : on a utilisé la sémantique particulière du `switch ... case` en `C`, attention à bien la comprendre !

Remarque 4 : cette écriture n'est pas nécessairement la plus efficace !

5.1.1.7 Correction d'une inexactitude

On signale alors que la règle utilisée pour le calcul des années bissextiles est inexacte : en effet, les années divisibles par 4000 et dont le quotient est impair (comme 4000, 12000...), ne sont pas bissextiles[3] ! Il suffit alors de modifier la fonction *bissextile*, pour la version suivante :

```
int bissextile (int annee)
{
  if (annee % 4)
    return 0;
  if (annee % 100)
    return 1;
  if (annee % 400)
    return 0;
  if (annee % 4000)
```

[3]Cette modification proposée par l'astronome Herschell n'a jamais été adoptée...

```
    return 1;
  if ((annee / 4000) % 2)
    return 0;
  return 1;
}
```

On voit alors qu'aucun autre code n'est modifié, en particulier, pas le code de Nj_1 et Nj_2 qui utilisent cette fonction. C'est aussi un avantage de cette conception : à partir du moment où la sémantique (ce qu'elle est censée faire) et l'interface (son nom, ses paramètres, son type de résultat) ne sont pas modifiées, il n'est pas nécessaire de modifier les fonctions utilisatrices.

5.1.1.8 Le bug du 19 janvier 2038

Si on teste le programme précédent, avec la date du 19 janvier 2038 à 3 heures 14 minutes et 8 secondes (temps universel), on obtient un temps négatif (pour être précis, on obtient -2147483648) ! Ceci est dû au fait que les entiers (`int`) sont classiquement codés sur 32 bits : or, à la date indiquée, le nombre de secondes est de 2^{31}, qui dans le codage sur 32 bits, correspond à $-2^{31} = -2147483648$.

Il faudra donc d'ici là trouver une parade à ce problème : si l'on se réfère au bug de l'an 2000, on peut estimer que la majorité du travail de correction sera effectuée en décembre 2037... Plus sérieusement, la parade est déjà trouvée : il suffit de passer à un codage des dates sur 64 bits (la majeure partie des systèmes dits « 64 bits » le font déjà). Ce codage ne posera problème que dans (approximativement) 292 milliards d'années.

Encore une fois, il faut être conscient des limitations de l'implantation informatique des fonctions.

5.1.2 Le problème des reines sur un échiquier

On cherche à placer N reines sur un échiquier NxN de façon à ce qu'aucune des reines ne soit en mesure de « prendre » une autre (deux reines sont toujours sur des horizontales, verticales et diagonales différentes).

Plus précisément, on cherche à écrire un programme dont le seul argument est la valeur de N : le programme doit alors afficher sous une forme dessinée (pas besoin de graphique compliqué, se contenter de dessins alphanumériques) les reines sur l'échiquier. En fin de programme, on indique le nombre de solutions différentes obtenues.

5.1.2.1 Un peu d'analyse

On peut résoudre ce problème de façon brutale (les anglo-saxons appellent cette méthode *brute force algorithm*) : on génère toutes les positions possibles des N reines sur l'échiquier, et on teste pour chaque position si les reines peuvent se prendre !

On peut toutefois améliorer un peu l'algorithme en remarquant que chaque ligne de l'échiquier doit contenir une et une seule reine. On peut alors représenter une position des N reines par un tableau de taille N, l'élément d'indice i du tableau correspondant au numéro de colonne dans laquelle se trouve la reine de la ligne.

L'algorithme peut alors s'écrire très simplement :

- générer la position « initiale », initialiser le nombre de solutions à zéro ;
- tant que l'on n'a pas fini
  - si la position est convenable, la dessiner et incrémenter le nombre de solutions (sinon, ne rien faire) ;
  - générer la position suivante ;
- afficher le nombre de solutions obtenues.

Comme d'habitude, nous utiliserons un tableau de taille maximale fixée, dont une partie seulement sera effectivement utilisée pour un N particulier (N inférieur à la taille maximale, bien entendu !).

5.1.2.2 Génération des positions

L'algorithme indiqué ci-dessus parle de position initiale et de position suivante : encore convient-il de préciser ce que sont ces notions !

Revenons à notre modèle : une position est pour nous un tableau de N entiers de 0 à N-1. On a donc un ordre naturel sur ces positions qui est l'ordre lexicographique : la première position est $(0, 0, \ldots, 0)$, la suivante est $(0, 0, \ldots, 1)$, jusqu'à $(0, \ldots, 0, N-1)$, puis $(0, \ldots, 1, 0)$, ...la dernière étant $(N-1, \ldots, N-1)$.

On peut très facilement générer la position initiale : il suffit de mettre à zéro tous les éléments du tableau ! Pour générer la position suivante, on incrémente le dernier élément du tableau. Si ce dernier élément prend la valeur N, alors on le remet à 0 est on incrémente l'avant dernier. Si cet avant-dernier élément prend la valeur N, on le remet à 0 et on incrémente l'antépénultième...Lorsque le premier élément prend la valeur N, on a fini !

Note : pensez au fonctionnement d'un compteur kilométrique, qui affiche des chiffres de 0 à 9...

On trouvera ci-dessous un premier morceau de notre programme.

```
#include <stdio.h>
/* taille maximale de l'echiquier */
#define NMAX 32

typedef int position[NMAX];

void init_position (int N,position p)
{
int i;
  for (i=0;i<N;i++)
    p[i]=0;
}

/* cette fonction rend 0 si on est a la derniere */
/* position, et 1 sinon */
int next_position (int N,position p)
{
int j;
  for (j=N-1;j>=0;j--)
    if (++p[j]==N)
     p[j]=0;
    else
     break;
  return j!=-1;
}

int main (int argc,char *argv[])
{
position pos;
int N,nb;
  if (argc==2)
    N=atoi(argv[1]);
  else
    N=8;
  init_position(N,pos);
  for (nb=0;;) {
    if (test_position(N,pos)==1) {
     dessine_position(N,pos);nb++;
    }
    if (next_position(N,pos)==0)
     break;
  }
```

```
  (void) printf("%d positions trouvees\n",nb);
  return 0;
}
```

5.1.2.3 Test d'une position

Pour tester une position, on doit vérifier que pour tout couple de reines, les positions correspondent à des lignes différentes (c'est vrai par construction !), à des colonnes différentes et à des diagonales différentes.

Pour tester si ce sont des colonnes différentes, il suffit de vérifier que les valeurs des éléments sont différentes. Pour tester si ce sont des diagonales différentes, il suffit de tester si la valeur absolue de la différence entre les numéros de lignes est différente de la valeur absolue de la différence entre les numéros des colonnes (faire un dessin, si besoin). On en déduit le code de la fonction de test :

```
int test_position (int N,position p)
{
int i,j;
  for (i=0;i<N;i++)
    for (j=0;j<i;j++)
     if (p[i]==p[j] ||
      (p[i]-i)==(p[j]-j) ||
      (p[i]+i)==(p[j]+j))
       return 0;
  return 1;
}
```

5.1.2.4 Dessin de l'échiquier

Laissé à titre d'exercice au lecteur...

5.1.2.5 Quelques résultats

On obtient les résultats suivants, pour les petites valeurs de N :
- N=1 1 position
- N=2 0 position
- N=3 0 position
- N=4 2 positions
- N=5 10 positions
- N=6 4 positions
- N=7 40 positions
- N=8 92 positions

On notera que le temps de calcul pour N=8 commence à être long : ceci s'explique par la complexité en N^N de l'algorithme, soit déjà plus de 16 millions de positions à tester ! Cette méthode est bien sûre loin d'être efficace...

5.1.3 Toujours plus complexe

On a vu dans les exemples précédents que la complexité d'un problème pouvait porter sur les traitements : les données sont « simples » mais les traitements à effectuer sur icelles sont à analyser, décomposer...

Il est un autre domaine de complexité portant cette fois-ci sur les données du problème : leur nombre, leur diversité, amènent à une analyse plus poussée de ces données, d'où peuvent en découler d'ailleurs des réflexions sur les traitements. C'est ce qui va être mis en évidence sur l'exemple suivant.

On veut écrire un programme permettant de jouer au *MasterMind*. L'ordinateur choisit une combinaison de couleurs, et l'utilisateur essaie de la deviner en proposant des combinaisons : pour chaque combinaison proposée, l'ordinateur indique le nombre de couleurs « bien placées » et le nombre de couleurs « mal placées ».

5.1.3.1 Description précise du jeu

Il convient tout d'abord de bien préciser le fonctionnement voulu du jeu, car bien que ce soit un jeu connu, un certain nombre de variantes existent !

La combinaison à découvrir est une suite (ordonnée) de m couleurs, ces couleurs étant prises parmi p couleurs disponibles. Selon le choix de l'utilisateur, chaque couleur peut être utilisée une seule fois (jeu simple), ou plusieurs fois (jeu plus complexe). On souhaite que l'utilisateur puisse choisir la valeur de m entre 2 et 6, et la valeur de p entre 2 et 8, les 8 couleurs (au maximum) utilisées étant rouge, vert, bleu, jaune, marron, violet, orange, rose. On considère également que le nombre de propositions est limité à M, variant de 6 à 12 : l'utilisateur a perdu s'il ne trouve pas la combinaison au bout des M essais.

Le programme choisit une combinaison aléatoirement (si possible, toutes les combinaisons ont une probabilité égale) ; ensuite, tant que l'utilisateur n'a pas trouvé la solution, le programme demande une nouvelle proposition. Il compare cette proposition à la combinaison à trouver, et indique combien de couleurs sont bien placées et combien sont mal placées (attention : si une couleur apparaît plusieurs fois dans la

combinaison ou dans la proposition, on ne doit la compter qu'une seule fois !). Si l'utilisateur a trouvé la combinaison, il indique le nombre de coups joués. Si le nombre maximal d'essais est atteint, le programme indique au joueur qu'il a perdu et donne la combinaison.

Enfin, pour aider l'utilisateur, le jeu doit proposer les aides suivantes :

- chaque nouvelle proposition valide est comparée aux précédentes et les incohérences sont signalées ;
- il doit être possible de revoir l'ensemble des propositions déjà faites avec les scores associés.

5.1.3.2 Analyse, modélisation des données

Paramètres Comme indiqué ci-dessus, on note m le nombre de couleurs de la combinaison, p le nombre de couleurs possibles, M le nombre maximal de combinaisons proposables par l'utilisateur. On a également besoin d'un indicateur permettant de savoir si une couleur peut être utilisée une seule fois ou plusieurs fois : on notera `multi` cet indicateur, valant 1 si l'on peut répéter une couleur, et 0 sinon.

Note : une fois choisis, ces paramètres ne sont plus modifiés au cours du jeu.

Couleurs On a bien sûr besoin de représenter informatiquement les couleurs ! Chaque couleur sera représentée par un entier entre 0 et $p-1$. Puisque la valeur maximale de p est 8, cela donne donc un entier entre 0 et 7. On peut donc utiliser des variables de type `char` (entier sur 8 bits) pour coder la couleur.

On a également besoin d'une représentation « textuelle » des couleurs (il est plus commode pour le joueur de lire `rouge`, `vert` ou `bleu` que 0, 3 ou 6...). On utilisera pour cela des chaînes de caractères, stockées dans un tableau : à l'indice i du tableau correspond la description textuelle de la couleur i.

Combinaison Comme une combinaison est une série de couleurs, on va utiliser un tableau de couleurs (donc, un tableau de `char`) pour coder les combinaisons.

Le nombre de couleurs de la combinaison étant limité à 6, tous les tableaux seront de taille 6 (on utilisera toutefois une *macro* en `C` pour définir cette valeur). Si le nombre m est inférieur à 6, on n'utilisera qu'une partie de chaque tableau.

Tableau de jeu Il est nécessaire de conserver en mémoire toutes les combinaisons jouées (afin de pouvoir gérer l'aide). Comme chaque combinaison est un tableau, on va donc utiliser un tableau à deux dimensions pour gérer ce tableau de jeu : le premier indice correspond au numéro de combinaison, le deuxième indice correspond au numéro de la couleur au sein de la combinaison.

On va utiliser un tableau de taille 12x6 (toujours grâce à une *macro*) dont on n'utilisera effectivement que la partie Mxm (le coin supérieur-gauche ou nord-ouest).

Scores On a besoin, pour chaque proposition, de stocker le « score ». Ce score étant ici composé de deux valeurs, on utilisera un tableau à deux dimensions, chaque ligne comportant deux valeurs : l'indice 0 correspond au nombre de bien placées, l'indice 1 correspond au nombre de mal placées. Encore une fois, le nombre de lignes du tableau sera pris comme une valeur maximale (nombre d'essais) dont seule une partie sera effectivement utilisée pour une valeur de M donnée.

Traduction en `C` Traduit en `C`, cela donne la version suivante :

```
#define MAXCOMBI 6
#define MAXCOULEURS 8
#define MAXESSAIS 12

typedef char mm_combi[MAXCOMBI];
typedef char mm_jeu[MAXESSAIS][MAXCOMBI];
typedef int mm_scores[MAXESSAIS][2];

int m,M,p,multi;
```

Note : on a utilisé la définition de type en `C` (mot-clé `typedef`) permettant de donner un nom de type « parlant » à nos variables.

5.1.3.3 Les traitements

Une fois les données modélisées, on peut s'intéresser aux différents traitements que le programme doit effectuer. Chaque traitement fera l'objet d'une (ou plusieurs) procédure qui réalise informatiquement le traitement correspondant. On précisera alors les données d'entrée et les résultats en sortie de chaque procédure.

5.1.3.4 Tirage au sort de la combinaison

Il s'agit de choisir aléatoirement m valeurs entre 0 et p, tirage effectué sans remise si on n'autorise qu'une seule fois chaque couleur, avec remise si on peut utiliser plusieurs fois chaque couleur.

On sait que l'on dispose d'une fonction `rand()` qui, d'après notre documentation, effectue un tirage pseudo-aléatoire entre 0 et `RAND_MAX`. Pour obtenir un tirage pseudo-aléatoire entre 0 et $n-1$, il suffit d'effectuer un modulo n (on négligera les imperfections [4] de répartition d'un tel tirage).

Si le tirage se fait avec remise, chaque couleur de la combinaison est obtenue par un tirage indépendant des autres. Si le tirage est fait sans remise, on peut utiliser une procédure ressemblant beaucoup à un exercice du chapitre 4 (génération d'une permutation aléatoire) en tronquant la génération après le p^{e}élément.

Les paramètres de cette procédure sont donc :
- en entrée, `m`, `p` et l'indicateur `multi` ;
- en sortie, la combinaison initiale ; comme tout résultat de type tableau, le paramètre est passé par adresse à la procédure.

d'où un prototype en `C` de cette procédure :

```
int init_combi (int m,int p,int multi,mm_combi la_combi);
```

La fonction rend 0 si le tirage a réussi, -1 en cas de problème (tirage sans remise avec m supérieur à p). On peut bien sûr décomposer cette fonction en deux, l'une pour le tirage sans remise, l'autre pour le tirage avec remise. Voici une façon d'implémenter ces deux fonctions :

```
int init_combi_remise (int m,int p,mm_combi la_combi)
{
int i;
  for (i=0;i<m;i++)
    la_combi[i]=(rand() % p);
  return 0;
}

int init_combi_sansremise (int m,int p,mm_combi la_combi)
{
char aux[MAXCOULEURS];
int i,j;
  if (m>p)
return -1;
  for (i=0;i<p;i++)
```

[4] Pourquoi ?

```
    aux[i]=i;
  for (i=0;i<m;i++) {
    j=(rand() % (p-i));
    la_combi[i]=aux[i+j];
    aux[i+j]=aux[i];
  }
  return 0;
}

int init_combi (int m,int p,int multi,mm_combi la_combi)
{
  srand(time(0));
  if (multi)
    return init_combi_remise(m,p,la_combi);
  return init_combi_sansremise(m,p,la_combi);
}
```

5.1.3.5 Lecture de la proposition de l'utilisateur

On souhaite que l'interface permettre à l'utilisateur de donner sa proposition sous la forme de noms de couleurs. On utilisera pour séparer les couleurs des caractères spéciaux comme les espaces, les tabulations, ou éventuellement les virgules. Sont ainsi des entrées convenables (dans le cas $m = 4$ et $p = 6$) :

```
rouge vert bleu jaune
rouge, vert     bleu, jaune
```

ainsi que

```
rouge, vert
bleu, jaune
```

On fera en sorte que, si le joueur entre une ligne vide ou un nombre de couleurs insuffisant, le programme affiche l'aide (impression des combinaisons précédemment jouées). On prendra également garde à ne pas lire trop de couleurs (on stoppe dès que l'on en a lu m), et à vérifier que les couleurs sont bien les p premières.

Notre fonction de lecture aura alors comme prototype :

```
int lire_la_combi (int m,int p,mm_combi la_combi);
```

la valeur retournée étant le nombre de couleurs effectivement lues.

Note : à ce niveau du cours, les entrées-sorties n'ont pas été abordées, et on ne détaillera pas l'implémentation de cette procédure.

Avertissement : on rappelle que les erreurs de conception des entrées-sorties sont fréquemment à l'origine de dysfonctionnements des programmes.

5.1.3.6 Manipulation des couleurs

On va avoir besoin pour imprimer les combinaisons, ou bien pour effectuer la saisie des combinaisons, de convertir le nom d'une couleur en son numéro et réciproquement.

Il est alors souhaitable d'écrire les deux fonctions de conversion suivantes :

```
int numero_couleur (char *str_couleur);
char *chaine_couleur (int num_couleur);
```

la première donnant le numéro de la couleur définie par la chaîne de caractères passée en argument (et -1 en cas d'erreur), la seconde donnant la description sous forme de chaîne de caractères de la couleur passée en argument (et 0 en cas d'erreur).

On trouvera ci-après une (ce n'est pas la seule) façon d'implémenter ces deux fonctions :

```
char *tab_couleurs[MAXCOULEURS] =
 {"rouge","vert","bleu","jaune",
 "marron","violet","orange","rose"};

int numero_couleur (char *str)
{
int i;
  for (i=0;i<MAXCOULEURS;i++)
    if (!strcmp(str,tab_couleurs[i]))
      return i;
  return -1;
}

char *chaine_couleur (int num)
{
  if ((num<0)||(num>=MAXCOULEURS))
    return 0;
  return tab_couleurs[num];
}
```

5.1.3.7 Calcul du score d'une proposition

Étant donné une combinaison et une proposition, on doit calculer le nombre de couleurs bien placées et le nombre de couleurs mal placées. Il faut bien prendre garde à ce qu'une couleur ne soit jamais comptée deux fois !

Pour cela, on commence par calculer le nombre de couleurs bien placées : on compare les couleurs aux mêmes indices dans la combinaison et dans la proposition. Si les couleurs sont identiques, on ajoute un au nombre de bien placées. Afin d'éviter de recompter cette couleur en tant que mal placée, on remplacera la couleur dans la combinaison par une valeur incohérente (`MAXCOULEURS`, par exemple) et dans la proposition par une valeur incohérente différente (`MAXCOULEURS+1`, par exemple). Il faut donc faire attention à faire des copies préalables de la combinaison et de la proposition !

On peut calculer ensuite les couleurs mal placées : il faut cette fois-ci tester les couleurs à des positions différentes dans la combinaison et dans la proposition. On utilisera le même mécanisme de remplacement des couleurs par des valeurs incohérentes si l'on trouve des couleurs concordantes. On obtient alors le code `C` suivant :

```
  /* le resultat est le nombre de "bien places" */
  int teste_combi (int m,mm_combi la_combi,mm_combi essai,
    int score[2])
  {
  mm_combi la_combibis,essaibis;
  int i,j;
    for (i=0;i<m;i++) {
      la_combibis[i]=la_combi[i];
      essaibis[i]=essai[i];
    }
    /* on calcule le nombre de bien places : on modifie
*/
    /* alors la combi et l'essai pour ne pas les */
    /* recompter en tant que mals places */
    for (i=0,score[0]=0;i<m;i++)
      if (la_combi[i]==essai[i]) {
       score[0]++;
       la_combibis[i]=MAXCOULEURS;
       essaibis[i]=MAXCOULEURS+1;
      }
    /* on calcule le nombre de mal places : on modifie */
    /* encore la combi et l'essai pour ne pas compter */
    /* deux fois la meme couleur */
```

```
    for (i=0,score[1]=0;i<m;i++)
      for (j=0;j<m;j++)
       if ((i!=j)&&(la_combibis[i]==essaibis[j])) {
        score[1]++;
        la_combibis[i]=MAXCOULEURS;
        essaibis[j]=MAXCOULEURS+1;
       }
    return score[0];
  }
```

Note : la fonction rend un résultat de type entier alors qu'on s'attend à ce qu'elle rende un score. Ceci est dû au fait qu'en C, il n'est pas possible qu'une fonction renvoie un résultat de type tableau : on passe donc le tableau (l'adresse du tableau pour être précis) en argument à la fonction. Le résultat est ici le nombre de bien placées : on teste si ce résultat est égal à m pour savoir si l'on a trouvé la combinaison.

5.1.3.8 Gestion de l'aide

L'aide comprend deux parties :

- si le joueur ne saisit pas une proposition complète, on affiche les propositions précédentes et les scores ;
- si le joueur saisit une proposition correcte, on commence par vérifier qu'elle est cohérente avec les propositions précédentes ; cela signifie que le score obtenu par chaque proposition précédente doit être identique au score que l'on aurait obtenu si la combinaison était identique à la nouvelle proposition.

Pour afficher une proposition, on peut tout à fait écrire une fonction spécifique, comme celle indiquée ci-après :

```
void print_combi (int m,mm_combi la_combi)
{
int i;
  for (i=0;i<m;i++)
    (void) printf("%s%c",chaine_couleur(la_combi[i]),
     (i<(m-1)) ? ',' : '\n');
}
```

Note : on utilise la fonction `chaine_couleur` définie précédemment ; noter également l'utilisation de l'expression `x ? y : z`, qui si x est vrai (non nul) vaut y, et z sinon.

On peut alors écrire la fonction d'affichage des combinaisons précédentes elle-même :

```
void print_aide (int m,int i,mm_jeu le_jeu,mm_score sc)
{
int j;
  for (j=0;j<i;j++) {
    (void) printf("Prop. %d :",j);
    print_combi(m,le_jeu[j]);
    (void) printf("\t%d bien places, %d mal places\n",
     sc[j][0],sc[j][1]);
  }
}
```

Pour la vérification de la cohérence par rapport aux propositions précédentes, voici une façon de programmer la fonction :

```
/* teste si la proposition i est coherente avec les */
/* precedentes : rend 1 s'il y a incoherence, 0 sinon */
int teste_incoherence (int m,int i,mm_jeu le_jeu,
  mm_score le_score)
{
int j,tmp_score[2];
  for (j=0;j<i;j++) {
    (void) teste_combi(m,le_jeu[i],le_jeu[j],tmp_score);
    if ((tmp_score[0]!=le_score[j][0])||
      (tmp_score[1]!=le_score[j][1])) {
     (void) printf("proposition incoherente"
     " avec reponse %d\n",j);
     return 1;
    }
  }
  return 0;
}
```

5.1.3.9 Le jeu complet

On peut maintenant combiner les fonctions précédentes pour obtenir le jeu. On va en fait écrire une première fonction qui décrit une seule phase de jeu :

- saisie de la proposition de l'utilisateur ;
- test de validité, affichage de l'aide si besoin ;
- test de cohérence, affichage de l'aide si besoin ;
- calcul et affichage du score de cette proposition.

La fonction rend 1 si l'utilisateur a trouvé la solution, et 0 sinon. Voici une implémentation de cette phase de jeu :

```
int phase_jeu (int m,int p,int i,mm_combi la_combi,
  mm_jeu le_jeu,mm_score le_score)
{
  for (;;) {
    if (lire_combi(m,p,le_jeu[i])!=m) {
      (void) printf("erreur de saisie !\n");
      print_aide(m,i,le_jeu,le_score);
      continue;
    }
    if (teste_incoherence(m,i,le_jeu,le_score)) {
      print_aide(m,i,le_jeu,le_score);
      continue;
    }
    break;
  }
  if (teste_combi(m,la_combi,le_jeu[i],le_score[i])==m) {
    (void) printf("combinaison trouvee en %d coups !\n",i);
    return 1;
  }
  (void) printf("%d bien places, %d mal places\n",
    le_score[i][0],le_score[i][1]);
  return 0;
}
```

On peut maintenant écrire le jeu complet :

- génération de la combinaison à trouver ;
- répétition d'une phase de jeu tant que le nombre d'essais est inférieur au maximum et que la combinaison n'est pas trouvée ;
- si le joueur n'a pas trouvé, affichage de la solution.

ce qui peut se programmer comme suit :

```
int partie (int m,int p,int multi,int M)
{
mm_combi la_combi;
mm_jeu le_jeu;
mm_score le_score;
int i;
  if (init_combi(m,p,multi,la_combi)==-1) {
    (void) printf("ERREUR !\n");
    return 2;
  }
  for (i=0;i<M;i++)
    if (phase_jeu(m,p,i,la_combi,le_jeu,le_score))
      return 1;
```

```
    (void) printf("vous n'avez pas trouve,"
                  " la combinaison etait:\n");
    print_combi(m,la_combi);
    return 0;
  }
```

5.2 À retenir

- avant tout choix d'algorithme, se poser la question de la finalité de l'application à réaliser
- lorsque c'est possible et utile, réfléchir longuement à l'algorithme et aux structures de données à mettre en œuvre
- décomposer le problème en sous-problèmes plus simples, que l'on programmera sous forme de procédures, et éventuellement de modules séparés
- écrire les programmes de manière claire, lisible et aérée, en privilégiant les commentaires significatifs ; exemple *a contrario* :

```
i=i+1; /* On ajoute 1 à i */
```

5.3 Travaux pratiques

5.3.1 Exercice 1

Faire l'analyse, la modélisation, la conception du mini-projet (décrit à l'annexe C page 121). On réfléchira notamment aux points suivants :

- comment représenter les données du problème ?
- comment représenter la solution du problème ?
- par quel algorithme peut-on trouver la solution du problème ?
- question subsidiaire : cet algorithme est-il efficace, peut-on l'améliorer ?

Deuxième partie

Annexes

Annexe A

Quelques aspects du matériel

A.1 Processeur, ou unité centrale

Unité de commande
- transfert des instructions depuis la mémoire
- décodage et exécution des instructions

Registres : mémoire très rapide située dans l'UC
- *adresse* : désignent un élément de la mémoire
- *donnée* : représentent une valeur
- *état* : représentent une partie de « l'état » du processeur

Unité arithmétique et logique
- décodage des fonctions
- opérations arithmétiques de base $+ \; - \; \times \div$
- opérations logiques, décalages $\vee \wedge \; \sim$

Performances
- liées à la fréquence d'horloge et à la nature du processeur
- mesurées en **MIPS**, Millions of Instructions Per Second

A.2 Mémoire

Mémoire centrale
- Volatile
- Technologie : circuits silicium
- Espace linéaire d'éléments identiques
- Éléments : des octets (byte) de 8 bits ;
- Un élément peut représenter 2^8 valeurs distinctes
- Désignés par les entiers de l'intervalle $[0, \, N-1]$
- Les éléments sont directement accessibles par la CPU
- Les temps d'accès sont très réduits (10^{-8} secondes)
- Capacité : 256 Mo ($\simeq 10^8$ octets)

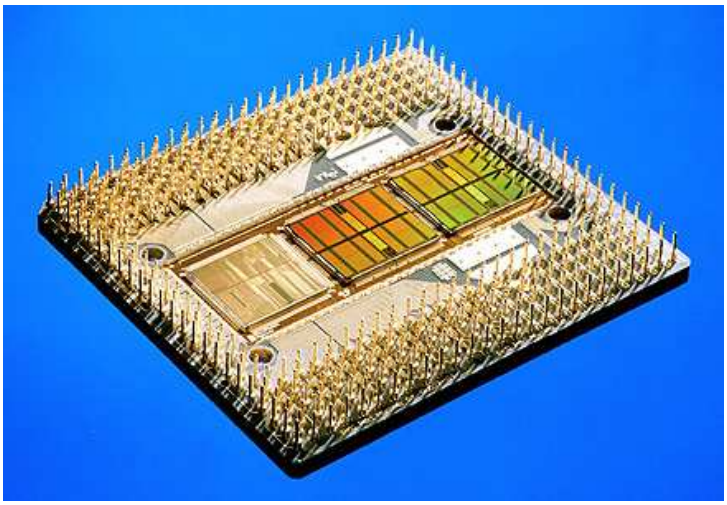

FIG. A.1 – Pentium Pro : processeur, 2 caches de 512 Ko

Mémoire secondaire

- Rémanente
- Diverses technologies (disques, CD ROMS...)
- Accès par blocs d'éléments
- Transferts vers (ou depuis) la mémoire centrale
- Temps d'accès plus importants (10^{-3} secondes)
- Capacité : 128 Go ($\simeq 10^{11}$ octets)

A.3 Bus

Un **bus** est un ensemble de « fils » connectant des unités fonctionnelles au sein d'un ordinateur

- bus interne CPU $\Leftrightarrow$ cache (300 bits Pentium Pro)
- bus donnée Processeur $\Leftrightarrow$ Mémoire
- lignes adresses [16, 32, 48 bits]
- lignes données [8, 16, 32, 64 bits]
- signaux [contrôle et logique]
- normes : ISA, PCI, etc...
- bus externe : Ordinateur $\Leftrightarrow$ Périphérique
- arbitrage : centralisé/décentralisé
- normes : IDE, SCSI, USB, etc...

A.4 Exemple de Processeur

La figure A.1 page précédente montre un Pentium Pro, processeur Intel sorti en mars 1995. Ses caractéristiques sont les suivantes :

- bus interne 300 bits, bus externe 64 bits
- mémoire 4 Go, mémoire virtuelle 64 To
- processeur : 5.5 M transistors, cache 62 M transistors
- horloge 200 MHz

Annexe B

Environnement de développement

Avant-propos

Il ne nous (les enseignants en informatique) a pas semblé souhaitable pour ce cours d'introduction à l'informatique de mettre à disposition des élèves un environnement de développement intégré tel que ceux que l'on peut trouver dans l'industrie. En effet, par leur côté intégré, ils tendent parfois à cacher les concepts informatiques (fichiers, processus, programme, compilateur, débogueur, etc.) sous-jacents, dont l'apprentissage est aussi un des objectifs du cours.

De plus, les environnements choisis (Linux et *gcc* d'un côté, et DJGPP sous Windows de l'autre) font partie de la grande famille des logiciels libres, et à ce titre sont utilisables dans tout contexte, y compris commercial. Un avantage non négligeable est aussi le fait qu'on y trouve un grand nombre d'utilitaires bien commodes pour les enseignants car identiques à ceux qu'ils utilisent quotidiennement.

Vous pourrez donc parfois trouver frustes ce mode de travail et ces environnements, mais ils conviennent tout-à-fait au but que nous cherchons à atteindre dans ce premier cours.

B.1 L'environnement de base

Rappel : le matériel utilisé est le parc de micro-ordinateurs du 4^{e}étage, utilisant au choix le système d'exploitation Microsoft Windows2000 ou le système Linux.

La fabrication de vos programmes va nécessiter l'utilisation successive de plusieurs outils : éditeur de texte, compilateur, débogueur. L'environnement de programmation n'étant pas intégré, vous aurez donc à

FIG. B.1 – Commande DOS

déclencher l'exécution de ces outils « manuellement ».

B.2 Utilisation sous Windows

Nous utilisons sous Windows un environnement de développement particulier, DJGPP, qui permet d'utiliser le compilateur `gcc` du projet GNU.

B.2.1 L'interpréteur de commandes

Pour mettre en œuvre DJGPP, il est nécessaire de disposer d'un interpréteur de commandes, donc le rôle est précisément de saisir des lignes de texte tapées au clavier pour déclencher l'exécution des commandes concernées. Sous l'environnement Windows, le plus connu de ces interpréteurs est la « boîte MS-DOS », que vous pouvez lancer par le menu `Démarrer` puis le sous-menu `Programmes`, puis le sous-sous-menu `Accessoires` (choix `Invite de commandes`)[1]. La figure B.1 montre une telle fenêtre. Il est possible également d'utiliser une fenêtre bash (c.f. figure B.2 page suivante), dont l'interprète utilise une syntaxe légèrement différente de celle de l'interprète de commandes DOS.

Le lancement d'un programme se fait par l'intermédiaire d'une *commande*, c'est-à-dire une ligne de texte frappée au clavier et terminée par la touche `Return` ; par exemple, la commande

[1] l'auriez-vous trouvé seuls ?

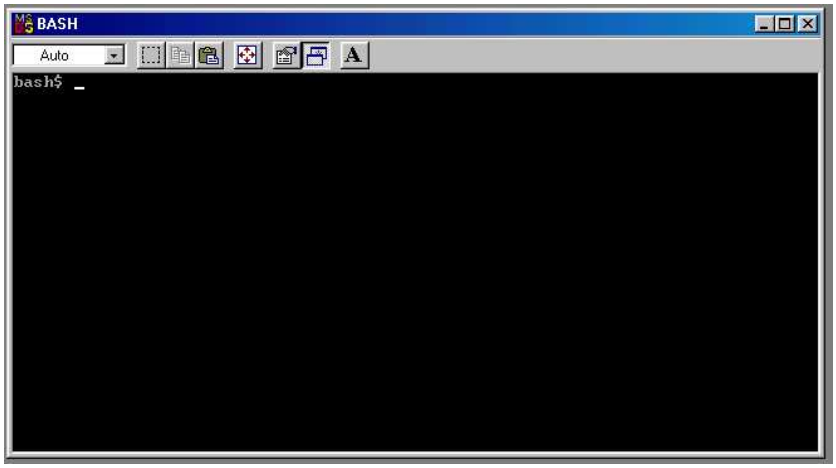

FIG. B.2 – Fenêtre *bash* sous DOS

```
C:\TMP> notepad
```

va cherche un fichier exécutable (dont le nom est `notepad.exe` ou `notepad.com`), et déclencher l'exécution de ce programme. Ces fichiers sont recherchés dans le répertoire courant et dans un ensemble de répertoires précisés par la commande DOS `path`.

Certaines commandes admettent des arguments et/ou des options, qui sont des chaînes de caractères qui suivent le nom du programme (N.B. : le programme est totalement maître du format et de la signification de ces arguments supplémentaires).

```
C:\TMP> notepad foo.c
C:\TMP> gcc -o foo.exe foo.c
```

Note : DJGPP fournit un autre interpréteur, nommé `bash`, dérivé du *Bourne shell* connu sur les stations Unix. C'est un interprète offrant de très nombreuses fonctionnalités (*wildcarding* en particulier).

Voici quelques commandes :

exit interrompre l'interaction

dir liste des fichiers du répertoire courant

. le répertoire courant

.. la racine du répertoire

cd changement de répertoire

```
c:> cd ..\durand\cours
```

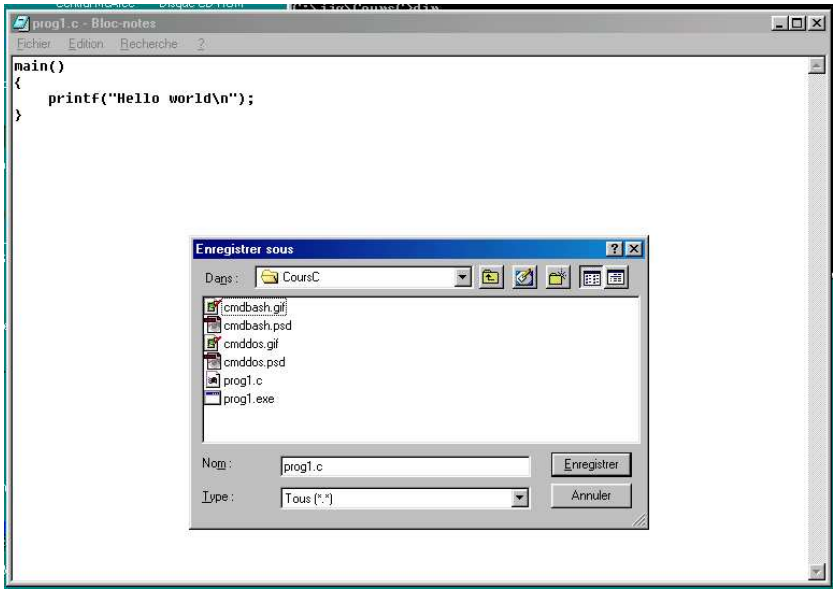

FIG. B.3 – Le *notepad* Windows et sa fenêtre de sélection de fichier

mkdir nom créer un nouveau répertoire

copy nom1 nom2 recopier un fichier

```
c:> copy prog2.c a:\durand
```

rem nom supprimer un fichier

rename nom1 nom2 renommer « nom1 »

```
c:> rename toto.c.txt toto.c
```

toto exécuter le programme `toto.exe` ou `toto.com`

```
c:> notepad
```

a : passer sur la disquette

c : passer sur le disque c :

B.2.2 L'éditeur de texte

Le premier programme que vous aurez à utiliser pour construire vos propres programmes est l'éditeur de texte : c'est lui qui va vous permettre d'écrire vos programmes source en langage `C`.

Vous pouvez utiliser l'un des éditeurs fournis avec le système :

– le *bloc-notes*, aussi appelé `notepad`, assez rudimentaire (c.f. figure B.3) ;
– l'éditeur `wordpad`, un peu plus sophistiqué.

Vous pouvez aussi utiliser tout autre éditeur de texte à votre convenance. Rappelez-vous simplement que :

- vos fichiers sources doivent être des fichiers en mode texte seulement ;
- vos fichiers sources en `C` **doivent** avoir un nom se terminant par `.c` : cette terminaison particulière permet au compilateur (voir plus loin) de deviner le type de langage contenu dans le fichier.

B.2.3 Le compilateur

Le programme permettant de fabriquer des programmes exécutables à partir de vos fichiers sources est `gcc` (*GNU C compiler*). C'est en fait un enchaîneur de passes, qui va successivement lancer :

- le préprocesseur (travaille uniquement au niveau du texte, aucune connaissance de la syntaxe du langage) ;
- le compilateur (traduit le code source en langage d'assemblage) ;
- l'assembleur (traduit le langage d'assemblage en code objet) ;
- l'éditeur de liens (ajoute au code objet les routines de démarrage et d'arrêt ainsi que les fonctions pré compilées nécessaires).

B.2.3.1 Compilation d'un programme indépendant

Si votre fichier source s'appelle `prog.c`, la fabrication du programme se fait par la commande :

```
gcc prog.c
```

qui construit (si tout se passe bien !) le programme nommé `a.exe`. Si l'on veut donner un autre nom au programme, on peut utiliser une option de `gcc` :

```
gcc -o prog.exe prog.c
```

l'option `-o` étant suivie de l'argument `prog.exe` qui est le nom à donner au programme résultat (encore une fois, le suffixe `.exe` permet de repérer le type du contenu du fichier).

Si vous souhaitez avoir des messages d'avertissement (qui sont des aides véritables à la mise au point lorsque l'on sait les interpréter), vous pouvez utiliser l'option `-W` du compilateur :

```
gcc -o prog.exe -Wall prog.c
```

Pour lancer le programme, il suffit alors de taper :

```
prog.exe
```

ou encore plus simple :

```
prog
```

B.2.3.2 Génération de fichiers intermédiaires

La fabrication d'un programme est une succession de passes. Il est possible de demander l'exécution des passes une par une, afin de générer les fichiers intermédiaires. Par exemple, le préprocesseur peut être appelé par :

```
gcc -o prog.i -E prog.c
```

puis le compilateur par

```
gcc -o prog.s -S -Wall prog.i
```

puis l'assembleur par

```
gcc -o prog.o -c prog.s
```

et enfin l'éditeur de liens par

```
gcc -o prog.exe prog.o
```

Les fichiers `prog.i` (pré-traité), `prog.s` (compilé) sont des fichiers texte que vous pouvez lire avec votre éditeur. Les fichiers `prog.o` (objet) et `prog.exe` (exécutable) sont des fichiers binaires.

B.2.4 Mise au point des programmes

Il est vraisemblable que certains de vos programmes comporteront des erreurs (des *bogues* en français, traduction des *bugs* anglais). Un outil très pratique pour détecter ces bogues est le metteur au point, ou *débogueur*, dont le rôle est d'exécuter sous contrôle le programme fautif et de pouvoir examiner le contenu de la mémoire, la succession des appels de fonctions, etc...

Lorsque le compilateur utilisé est `gcc`, l'outil de mise au point à utiliser est `gdb`.

B.2.4.1 La préparation du débogage

Pour ce faire, le débogueur a besoin d'une version particulière du programme exécutable : cette version est obtenue par une option spécifique du compilateur. Ainsi, la construction d'un programme en vue de son débogage s'obtient par

```
gcc -o prog -g prog.c
```

Notez :

– l'utilisation de l'option `-g` (si l'on compile passe par passe, il faut préciser cette option pour la phase de compilation et la phase d'édition de liens).

B.2.4.2 Mise au point sous Windows

Le débogueur `gdb` est fourni avec l'environnement DJGPP.

On notera que, lors de la compilation, le nom passé en argument de l'option -o ne doit pas comporter le suffixe .exe.

Le compilateur construit deux fichiers : l'exécutable classique `prog.exe` un autre fichier `prog`, ce dernier étant requis par le débogueur.

B.2.4.3 Mise au point sous Linux

Le débogueur de choix sous Linux est `gdb`.

B.2.4.4 Description de gdb

`gdb` permet de déboguer du `C`, du `C++` et du `Modula2`. On peut aussi l'étendre à d'autres langages. C'est un débogueur en ligne de commande : lors de son exécution, il affiche un petit message (le *prompt*, habituellement « `(gdb)` ») et attend les commandes de débogage. Il exécute la commande, affiche éventuellement le résultat, et attend la commande suivante.

Pour appeler ce débogueur sur le programme `prog` (attention : ne pas oublier la procédure particulière pour générer un programme en vue d'un débogage), il suffit de lancer la commande :

```
gdb prog
```

B.2.4.5 Un premier exemple

Voici un programme dont l'exécution va entraîner une division par zéro :

```
#include <stdio.h>
int main(int argc, char * argv[])
{
    int tab[10], i;
    for (i=0; i<10; i++)
        tab[i] = (i+3)/(i-5);
    printf("Fini\n");
}
```

La compilation et l'exécution nous donnent :

```
[CTEST]$ gcc -g bug1.c
[CTEST]$ a.out
Floating point exception
```

```
[CTEST]$
```

Le lancement de ce programme sous le metteur au point permet d'avoir une idée précise de la nature et de l'emplacement de l'erreur (les commandes tapées par l'utilisateur sont indiquées en **gras**) :

```
[CTEST]$ gdb a.out
GNU gdb 5.2.1-2mdk (Mandrake Linux)
Copyright 2002 Free Software Foundation, Inc.
(gdb) run
Starting program : /home/girardot/CTEST/a.out
Program received signal SIGFPE, Arithmetic exception.
0x08048372 in main (argc=1, argv=0xbffff6a4) at bug1.c :8
8            tab[i] = (i+3)/(i-5);
(gdb) quit
The program is running. Exit anyway? (y or n) y
[CTEST]$
```

Notes : La commande « `gdb a.out` » est le lancement du débogueur sous le système d'exploitation utilisé (fenêtre de commande Windows ou Linux). Le débogueur est dès lors en attente de commande (son prompt est « `(gdb)` »).

Ici, l'exécution du programme à tester est déclenchée immédiatement par la commande « `run` ». L'erreur (une division par 0) est détectée à la ligne 8 du programme bug1.c, et le texte source de cette ligne est imprimée.

La commande « `quit` » permet, après confirmation, d'arrêt l'exécution du débogueur.

B.2.4.6 Un deuxième exemple

Voici un programme dont le temps d'exécution est très long, et donne l'impression qu'il a entamé une « boucle sans fin » :

```
#include <stdio.h>
void toto(int * d, int s)
{
    * d = s + * d;
    return;
}
int main(int argc, char * argv [])
{
    int i, j, k;
    i=j=k=0;
```

```
        for (i=0; i>=0; i++)
        {
            for (j=i; j>=0; j++)
                toto(&k,j);
        }
        printf("%d %d %d\n", i, j, k);
    }
```

L'exécution du programme ne donne aucun résultat visible, et au bout de quelques minutes, grande est la tentation de faire « contrôle-C » pour l'interrompre, ce qui, bien entendu, ne donne que peu d'indications sur son fonctionnement.

Voici le programme exécuté sous le contrôle du débogueur :

```
[CTEST]$ gdb bug2
GNU gdb 5.2.1-2mdk (Mandrake Linux)
Copyright 2002 Free Software Foundation, Inc.
(gdb) run
Starting program : /home/girardot/CTEST/bug2
Program received signal SIGINT, Interrupt.
0x08048396 in main (argc=1, argv=0xbffff6a4) at bug2.c :17
17              toto(&k,j);
(gdb)
```

Après avoir lancé le débogueur, puis l'exécution du programme à tester, nous avons au bout de quelques minutes tapé « contrôle-C » pour l'interrompre. `gdb` nous signale que cette interruption a eu lieu à la ligne 17 du programme source, avant l'appel de la procédure `toto`.

Nous pouvons consulter les valeurs des variables :

```
(gdb) print i
$1 = 1
(gdb) print j
$2 = 1832284597
(gdb) print k
$3 = -1345098718
(gdb)
```

Nous pouvons continuer l'exécution du programme interrompu :

```
(gdb) continue
Continuing.
```

Après une nouvelle interruption (par « contrôle-C »), nous pouvons consulter à nouveau les variables :

```
(gdb) print i
$4 = 6
```

L'évolution de la valeur de la variable « i » montre que le programme semble s'exécuter normalement. Nous pouvons exécuter la procédure courante ligne par ligne :

```
(gdb) next
16          for (j=i; j>=0; j++)
(gdb) next
17              toto(&k,j);
(gdb) next
16            for (j=i; j>=0; j++)
(gdb) next
17              toto(&k,j);
(gdb) print j
$7 = 1804878402
(gdb) next
16          for (j=i; j>=0; j++)
(gdb) next
17              toto(&k,j);
(gdb) print j
$8 = 1804878403
(gdb)
```

L'exécution en mode ligne par ligne montre l'enchaînement des appels de la fonction « `toto` » au sein de la boucle dans laquelle on fait varier la valeur de « `j` ».

Il est possible de suivre plus finement l'exécution au moyen de la commande « step ». Celle-ci va non seulement exécuter ligne par ligne, mais va de plus permettre de tracer les procédure appelées :

```
(gdb) step
toto (d=0xbffff63c, s=1804878403) at bug2.c :5
5       * d = s + * d;
(gdb) step
7   }
(gdb) step
main (argc=1, argv=0xbffff6a4) at bug2.c :16
16          for (j=i; j>=0; j++)
(gdb) step
17              toto(&k,j);
(gdb) step
toto (d=0xbffff63c, s=1804878404) at bug2.c :5
5     * d = s + * d;
(gdb) step
7   }
(gdb) step
```

```
main (argc=1, argv=0xbffff6a4) at bug2.c :16
16          for (j=i ; j>=0 ; j++)
(gdb) step
17              toto(&k,j);
(gdb) step
toto (d=0xbffff63c, s=1804878405) at bug2.c :5
5     * d = s + * d ;
(gdb) backtrace
#0 toto (d=0xbffff63c, s=1804878406) at bug2.c :5
#1 0x0804839f in main (argc=134513486, argv=0x1) at bug2.c :17
#2 0x4003b082 in __libc_start_main () from /lib/i686/libc.so.6
(gdb)
```

Notons enfin l'utilisation de la commande « `backtrace` » qui permet de connaître le contexte de la ligne en cours : nous sommes ici dans la procédure « `toto` » (ligne 5 du texte source « `bug2.c` »), appelée depuis la procédure « `main` » (à la ligne 17 du texte source « `bug2.c` »), elle-même appelée par la procédure générique de lancement d'un programme C, qui porte le nom barbare de « `__libc_start_main` ».

B.2.4.7 Récapitulatif de gdb

Commandes de base Pour lancer l'exécution du programme, il suffit de taper la commande :

```
(gdb) run
```

Si votre programme nécessite des arguments, on tape la commande :

```
(gdb) run arg1 arg2 ...
```

Il est possible d'utiliser pour cette commande « `run` » l'abréviation « `r` ».

Pour quitter le débogueur, on tape la commande :

```
(gdb) quit
```

Il est possible d'utiliser pour cette commande « `quit` » l'abréviation « `q` ».

On peut interrompre le programme courant en cours d'exécution par un « contrôle-C ».

À tout moment, un programme interrompu peut être continué par la commande « `continue` », dont l'abréviation est « `c` ».

B.2.4.8 Points d'arrêt

Un point d'arrêt permet de stopper l'exécution du programme à un endroit précis du code.

Pour mettre un point d'arrêt à l'entrée de la fonction `fonc`, il suffit de taper la commande :

```
(gdb) break fonc
```

Ainsi, « `break main` » permet d'interrompre le programme immédiatement avant le début de l'exécution de la procédure principale.

Pour stopper à une ligne déterminée du programme source (la ligne 14, par exemple), on utilise la commande

```
(gdb) break 14
```

Il est possible d'utiliser pour la commande « `break` » l'abréviation « `b` ».

On peut continuer l'exécution par la commande `continue` (abréviation « `c` »), ou bien exécuter en mode pas à pas par les commandes `step` ou `next` (abréviations « `s` » et « `n` » respectivement). On notera qu'il existe des instructions « `stepi` » et « `nexti` », similaires à « `step` » et « `next` », mais qui n'exécutent qu'une instruction de la machine à la fois.

Enfin, la commande `finish` permet de poursuivre l'exécution d'une procédure jusqu'à son retour à la procédure appelante, et d'interrompre l'exécution à ce moment.

B.2.4.9 Visualisation de données

Pour visualiser le contenu d'une variable, il suffit de taper

```
(gdb) print x
$1 = 4
(gdb) print tab[2]
$2 = 6.13
```

L'argument de la commande `print` est en fait toute expression `C` qui a un sens dans le contexte d'exécution courant.

B.2.4.10 Visualisation de code

Lors d'une exécution de programme, la commande `list` (abréviation « `l` ») permet d'afficher à l'écran les 10 lignes qui entourent la ligne courante.

B.2.4.11 La documentation en ligne

Le débogueur fournit la commande « `help` », qui permet de disposer d'indications précises sur la syntaxe d'une commande ou d'un groupe de commandes.

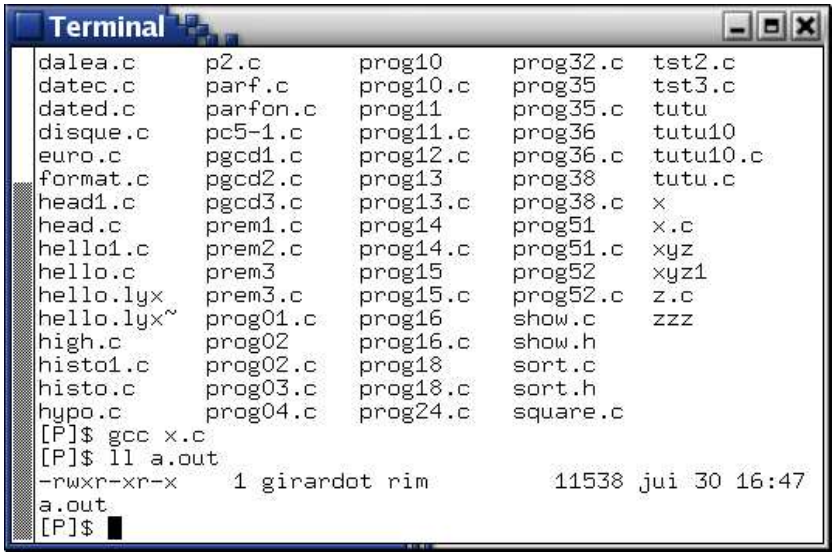

FIG. B.4 – Fenêtre `Xterm`

Sous Linux, la commande « `man` » fournit une indication plus ou moins détaillée sur le fonctionnement et les options des outils du système. Essayez par exemple « `man gcc` » ou « `man gdb` ».

DJGPP, ainsi que Linux, possèdent un système de documentation en ligne : on peut visualiser cette documentation par la commande `info`. On peut ainsi trouver de la documentation sur :

– le compilateur : `info gcc`
– le débogueur : `info gdb`

B.3 Utilisation sous Linux

Il n'y a pas d'outil particuliers à installer sous Linux, qui propose en standard le compilateur gcc, et nombre d'interprètes de commande et d'éditeurs de textes.

B.3.1 Fenêtre de commande

Les fenêtres de commandes se lancent par le choix d'un élément dans le menu « `Terminaux` ». Les figures B.4 et B.5 page suivante fournissent des exemples de l'apparence de tels terminaux. Le choix de l'interprète est laissé à l'utilisateur. Nous suggérons `bash` qui est l'un des plus répandus et offre diverses facilités de rappel et d'édition des commandes.

Quelques exemples de commandes :

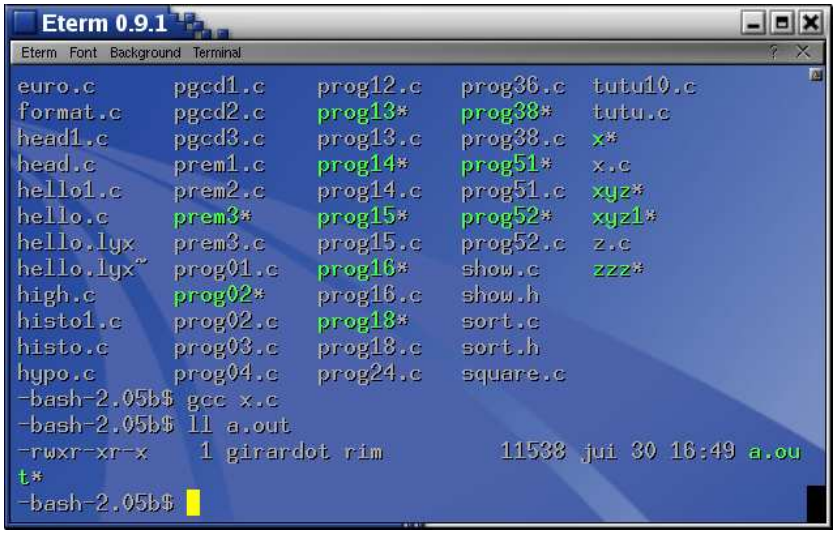

FIG. B.5 – Fenêtre `Eterm`

exit interrompre l'interaction

ls liste des fichiers du répertoire courant

- **.** le répertoire courant
- **..** la racine du répertoire

cd changement de répertoire

```
[tp] cd ../durand/cours
```

mkdir nom créer un nouveau répertoire

cp nom1 nom2 recopier un fichier

```
[tp] cp prog2.c /home/eleves/durand
```

rm nom supprimer un fichier

mv nom1 nom2 renommer « nom1 »

```
[tp] mv toto.c truc.c
```

toto exécuter le programme `toto`

```
[tp] toto
[tp] gcc -o titi titi.c
```

cd revenir dans son répertoire d'accueil

cd coursc/tp passer dans le répertoire désigné

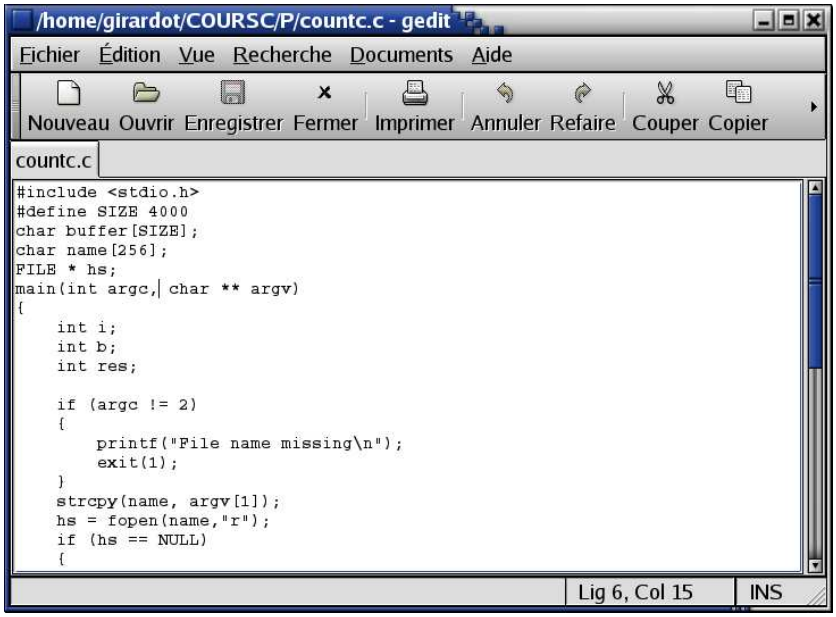

FIG. B.6 – L'éditeur *Gedit* sous Linux

B.3.2 Éditeur de textes

De nombreux éditeurs de textes sont disponibles, des plus simples, tels **`Gedit`** ou **`KWrite`**, aux plus complexes, tels **`vi`** ou **`emacs`**. **`Gedit`** (c.f. B.6) est un bon choix initial. Il offre une édition pleine page, l'utilisation de la souris, les copier-coller, etc. Il existe un menu spécialisé (Applications ▷ Éditeurs de texte) qui permet le lancement de ces outils, dont l'exécution peut être également demandée depuis la ligne de commande.

L'éditeur **`KWrite`** (c.f. B.7 page suivante) propose des fonctions similaires, et assure la coloration syntaxique des programmes, ce qui aide à la détection des apostrophes et parenthèses mal fermées. L'éditeur **`Kate`** est assez similaire.

Nous ne conseillons pas, dans un premier temps, l'utilisation des outils plus complexes, tels **`vi`** ou **`emacs`**, qui ont, bien sûr, leur raison d'être dans le cadre de développements de grande envergure.

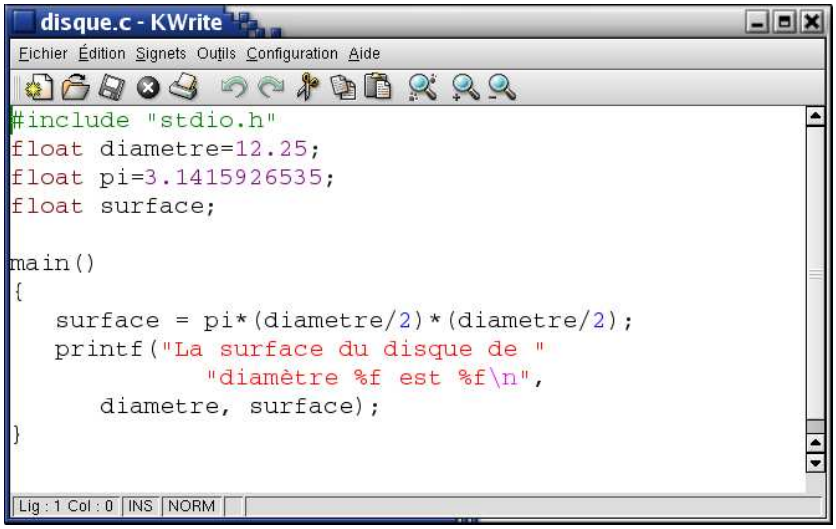

FIG. B.7 – L'éditeur *KWrite* sous Linux

Annexe C

Projet

C.1 Les règles du jeu

Afin d'assurer l'assimilation des concepts introduits lors des cours et travaux dirigés, il est proposé aux élèves de résoudre un petit problème par l'écriture d'un programme en langage `C`.

Ce programme est à réaliser **individuellement** par chaque élève (des programmes par trop ressemblants pourraient provoquer une certaine irritation des correcteurs). Il est à rendre en fin de semaine bloquée, par courrier électronique, en respectant **scrupuleusement** les indications suivantes :

- le courrier doit être arrivé au responsable à l'adresse `girardot@emse.fr` le vendredi soir (17 septembre pour le groupe A, 24 septembre pour le groupe B) à 18h30 ;
- le sujet du message doit être `Mini-projet C, groupe A/B` ;
- il doit être joint au message un attachement (fichier joint) qui est le code **source** du programme ; ce programme devra porter le nom de *login* de l'élève (`mroelens.c` par exemple) ;
- le corps du message pourra comporter quelques indications ou commentaires sur la réalisation du programme.

Chaque programme se verra appliquer le traitement suivant :

- tentative de compilation avec tous les warnings possibles (option `-Wall` de `gcc`) ; un warning non traité provoquera une diminution de l'évaluation ; un programme qui ne pourrait être compilé serait fort mal apprécié !
- tentative d'exécution ; un bouclage ou un plantage du programme serait encore fort mal apprécié ; la matérialisation d'un résultat (sous une forme opportune) sera appréciée positivement ; la justesse du ou des résultats sera appréciée encore plus positivement ;
- lecture attentive du code source afin d'évaluer la qualité de ré-

daction du programme ; un programme ne comportant pas de découpage opportun en fonctions serait fort mal ressenti, de même qu'un programme sans aucun commentaire !

C.2 Le problème

On veut réaliser une implémentation du célèbre *jeu de la vie* [1]. Dans ce jeu, on dispose d'un tableau rectangulaire de cellules : chaque cellule, qui a 8 voisines, peut être vivante ou morte. À partir d'un *motif* initial (qui indique les cellules vivantes et mortes initiales), on applique les règles de transformation suivantes :

- si à l'instant t, une cellule morte a exactement trois voisines vivantes, alors à l'instant $t + 1$, cette cellule est vivante (règle de naissance) ;
- si à l'instant t, une cellule vivante a deux ou trois voisines vivantes, alors à l'instant $t + 1$, cette cellule est vivante (règle de survie) ;
- si à l'instant t, une cellule ne vérifie aucune des deux conditions précédentes, alors à l'instant $t + 1$, cette cellule est morte.

Le but du projet est, à partir d'un motif initial, de construire et d'afficher le tableau de cellules à chaque étape.

C.3 Quelques précisions

C.3.1 Taille du tableau

Nous nous contenterons dans un premier temps de travailler sur un tableau rectangulaire fini, comprenant `NB_LIGNES` lignes de `NB_COLONNES` chacune (ces deux valeurs seront précisées dans le code source par des *macros*).

Pour les relations de voisinage aux bords de ce tableau, on pourra, au choix :

- soit ajouter une bordure fictive faisant le tour du tableau qui contient des cellules *toujours* mortes ;
- soit « replier » le tableau en considérant première et dernière ligne comme voisines, ainsi que première et dernière colonne.

C.3.2 Affichage

Pour réaliser l'affichage, on pourra, au choix :

[1] Voir par exemple `http ://www.math.com/students/wonders/life/life.html`

– soit utiliser la bibliothèque `curses` qui permet de gérer un terminal de type alphanumérique en mode plein écran ;
– soit utiliser la bibliothèque `vogle` qui permet de créer simplement une fenêtre de taille donnée, et d'afficher pixel par pixel dans cette fenêtre.

Vous trouverez sur le site du pôle `http://kiwi.emse.fr/POLE/` des exemples simples (fichiers `curs_ex.c` et `vogle_ex.c`) d'utilisation de ces bibliothèques à partir desquels vous pouvez construire votre propre programme.

C.3.3 Saisie du motif initial

La réalisation minimale comprend une procédure spécifique d'initialisation du motif (motif codé « en dur » dans le programme). Une réalisation un peu plus poussée permet de définir le motif à partir d'un ou plusieurs arguments de la ligne de commande (selon des modalités que vous devez définir).

Enfin, **si les étapes précédentes ont été implémentées avec succès**, il est possible de réaliser une saisie graphique à la souris (s'inspirer du programme d'exemple `vogle_ex2.c`).

Bibliographie

[1] Jean-Jacques Girardot et Marc Roelens. *Structures de données et Algorithmes en C*. `http://kiwi.emse.fr/CoursC`, septembre 2004.

[2] Christian Rolland. *LaTeX2ε, guide pratique*. Addison-Wesley, France, Juin 1995.

[3] LyX-Team. *Implémentation de LyX*. `http://www.lyx.org`.

Table des figures

Index

Table des matières