Université du Luxembourg

2005–2006

Cours de programmation avancée.

Le langage C

Sébastien Varrette <Sebastien.Varrette@imag.fr>

Nicolas Bernard <n.bernard@lafraze.net>

Version : 0.4

Table des matières

Avant-propos

Ce polycopié vient en complément au cours de programmation avancée dispensé au sein de l'université du Luxembourg en DUT d'informatique. En aucun cas il ne dispense de la présence en cours. Ce document n'a pas pour vocation d'être un ouvrage de référence du Langage C (pour cela, voir par exemple [KR88]). Il doit simplement permettre au lecteur d'appréhender rapidement ce langage.
Pour la petite histoire, le langage de programmation C a été développé dans les années 70 par Dennis Ritchie aux laboratoires Bell. Il puise ses origines dans le langage de programmation sans type BCPL (Basic Combined Programming Language, developpé par M. Richards) et dans B (developpé par K. Thompson). En 1978, Brian Kernighan et Dennis Ritchie produisent la première description officielle de C.
Le design de C lui confère un certain nombre d'avantages :
– Le code source est hautement portable
– Le code machine est très efficace
– les compilateurs C sont disponibles dans tous les systèmes courants.
La version courante de ce document ainsi que le code source des exemples abordés dans ce polycopié sont disponibles sur mon site
`http://www-id.imag.fr/~svarrett/`

Chapitre 1

Les bases

1.1 Un langage compilé

1.1.1 Généralités sur les langages de programmation

Dans le domaine de la programmation, on utilise aujourd'hui des langages de programmation de haut niveau (i.e "facilement" compréhensibles par le programmeur) par opposition aux langages de bas niveau (de type assembleur) qui sont plus orientés vers le langage machine. Parmi les exemples de langages de haut niveau, on peut citer les langages C, C++, Java, Cobol etc...
Il est néanmoins nécessaire d'opérer une opération de traduction du langage de haut niveau vers le langage machine (binaire). On distingue deux types de traducteurs :

1. l'*interpréteur* qui traduit les programmes instruction par instruction dans le cadre d'une intéraction continue avec l'utilisateur. Ainsi, le programme est traduit *à chaque exécution*.
2. le *compilateur* qui traduit les programmes *dans leur ensemble* : tout le programme doit être fourni en bloc au compilateur pour la traduction. Il est traduit une seule fois.

Il en résulte deux grands types de langages de programmation :
– les langages *interprétés* : dans ce cas, l'exécution des programmes se fait sous un "terminal" à la volée. Ce type de langage est donc adapté au développement rapide de prototypes (on peut tester immédiatement ce que l'on est en train de faire) On citera par exemple les langages **Scheme**, **Ruby**, etc...
– les langages *compilés* : les source du code est dans un premier temps passé dans une "moulinette" (le compilateur) qui va générer, sauf erreur, un exécutable qui est un fichier binaire compréhensible par votre machine. Ce comportement est illustré dans la figure 1.1. Ce type de langage est donc adapté à la réalisation d'applications plus efficaces ou de plus grande envergure (il y a une optimisation plus globale et la traduction est effectuée une seule fois et non à chaque exécution).
En outre, un langage compilé permet de diffuser les programmes sous forme binaire, sans pour autant imposer la diffusion le source du code, permettant

ainsi une certaine protection de la propriété intellectuelle. Quelques exemples de langages compilés : **C**, **C++**, **Java** etc...

A noter que certains langages peuvent être à la fois compilés et interprétés, comme par exemple **CAML**.

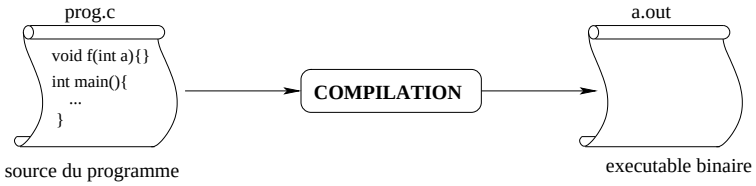


FIG. 1.1 – Compilation d'un code source

1.1.2 Le C comme langage compilé

Le langage C est donc un langage compilé (même s'il existe des interpréteurs plus ou moins expérimentaux). Ainsi, un programme C est décrit par un fichier texte, appelé *fichier source*. La compilation se décompose en fait en 4 phases successives[1] :

1. *Le traitement par le préprocesseur* : le fichier source (portant l'extension `.c`) est analysé par le préprocesseur qui effectue des transformations purement textuelles (remplacement de chaînes de caractères, inclusion d'autres fichiers sources, compilation conditionnelle ...). Les différentes commandes qui peuvent être fournies au préprocesseur seront exposées dans le chapitre 7.
2. *La compilation* : la compilation proprement dite traduit le fichier généré par le préprocesseur (d'extension `.i`) en assembleur, c'est-à-dire en une suite d'instructions du microprocesseur qui utilisent des mnémoniques rendant la lecture possible.
3. *L'assemblage* : cette opération transforme le code assembleur (extension `.s`) en un fichier binaire, c'est-à-dire en instructions directement compréhensibles par le processeur. Généralement, la compilation et l'assemblage se font dans la foulée, sauf si l'on spécifie explicitement que l'on veut le code assembleur. Le fichier produit par l'assemblage est appelé fichier objet (et porte l'extension `.o`). Les fichiers objets correspondant aux librairies pré-compilées ont pour suffixe `.a`.
4. *L'édition de liens* : un programme est souvent séparé en plusieurs fichiers sources (voir le chapitre 8 consacré à la programmation modulaire), pour des raisons de clarté mais aussi parce qu'il fait généralement appel à des librairies de fonctions standards déjà écrites. Une fois chaque fichier de code source assemblé, il faut donc lier entre eux les différents fichiers objets. L'édition de liens produit alors un fichier dit exécutable (`a.out` par défaut).

[1] Les extensions mentionnées peuvent varier selon le compilateur utilisé (`gcc` ici).

1.1.3 Notes sur la normalisation du langage C

[KR88], publié initialement en 1978 par Brian W. Kernighan et Denis M. Ritchie a fait office de norme pendant longtemps (on parle alors de C K&R, en référence aux 2 auteurs).
Devant la popularité du C, l'American National Standard Institut[2] charge en 1983 le comité X3J11 de standardiser le langage C. Le fruit de ce travail est finalement été approuvé en décembre 1989, le standard ANSI C89 est né.
L'International Organization for Standardization[3] a adopté en 1990 ce standard en tant que standard international sous le nom de ISO C90. Ce standard ISO remplace le précédent standard ANSI (C89) même à l'intérieur des USA, car rappelons-le, ANSI est une organisation nationale, et non internationale comme l'ISO.
Les standards ISO, en tant que tel, sont sujets à des révisions, notamment en 1995 (C95) et en 99 (on parle alors de C99). Comme on le verra dans la suite, certaines fonctionnalités décrites dans ce document héritent de la norme C99 et n'était pas présentes avant.

1.1.4 C en pratique : compilation et debuggage

Compilation d'un programme C

Il existe évidemment un grand nombre de compilateurs C pour chaque plateforme (Windows, Linux, MAC etc...). Le compilateur détaillé dans cette section correspond au compilateur libre `gcc`, disponible sur quasiment toutes les plateformes UNIX.
La compilation du fichier `prog.c` se déroule en général en deux étapes :

1. traduction du langage C dans le langage de la machine (binaire) sous forme d'un fichier objet `prog.o` :

   ```
   gcc -O3 -Wall -c prog.c
   ```

2. regroupement des variables et des fonctions du programme avec les fonctions fournies par des bibliotheques du systeme ou faites par l'utilisateur (edition de liens) : `gcc -o prog prog.o`

Ces deux commandes ont donc permis de générer l'exécutable `prog` qui peut alors est lancé (`./prog` dans une fenêtre de shell).
Si un seul fichier source est utilisé pour générer l'exécutable, les deux commandes précédentes peuvent être regroupées en une seule :

```
gcc -O3 -Wall prog.c -o prog
```

Quelques explications sur les options de `gcc` s'imposent :
– `-O` : effectue des optimisations sur le code généré. La compilation peut alors prendre plus de temps et utiliser plus de mémoire. Un niveau, variant de 0 à 3 spécifie le degré d'optimisation demandé. Ainsi, l'option `-O3` permet d'otenir le code le plus optimisé.
– `-Wall` : active des messages d'avertissements (warnings) supplémentaires ;
– `-c` : demande au compilateur de ne faire que la premiere étape de compilation ;

[2] `www.ansi.org`
[3] `www.iso.ch`

– `-o` : permet de préciser le nom du fichier produit.

il est important de noter que l'option `-Wall` est **indispensable** puisqu'elle permet de detecter la plupart des erreurs d'inattention, de mauvaise pratique du langage ou d'usage involontaire de conversion implicite.

Il existe bien sûr de nombreuses autres options disponibles[4]. En voici certaines qui pourront être utiles :

– `-Idir` : ajoute le répertoire `dir` dans la liste des répertoires ou se trouvent des fichiers de header `.h` (voir chapitre 8). Exemple : `-Iinclude/`
– `-Werror` : considère les messages d'avertissements (warnings) comme des erreurs
– `-g` : est nécessaire pour l'utilisation d'un debugger. Il est possible de fournir plus ou moins d'informations au debugger. Ce critère est caractérisé par un niveau variant entre 1 et 3 (2 par défaut). Ainsi, l'utilisation du niveau maximal (option `-g3`) assure que le maximum d'informations sera fourni au debugger ;
– `-pg` : nécessaire pour l'utilisation du profiler GNU : `gprof`[5] (qui permet de déterminer par exemple quelles sont les parties du code qui prennent le plus de temps d'exécution et qui doivent donc être optimisées).

Enfin, il est possible d'automatiser la compilation de l'ensemble des fichiers d'un projet à l'aide de l'utilitaire `make` qui sera présenté au §8.5.

Note sur les erreurs de compilation : A ce stade, on peut classer les erreurs de compilation en deux categories :

– les erreurs de syntaxe produites lors de la première phase lorsque le code n'est pas écrit en C correct.
– les erreurs de resolution des symboles produites lors de la deuxième phase lorsque des variables ou des fonctions utilisées ne sont definies nulle part.

Outils de debuggage

Il existe un certain nombre de debugger qui peuvent être utilisés pour corriger les erreurs d'un programme :

– en ligne de commande : `gdb` (`http://www.gnu.org/software/gdb/gdb.html`)
– sous forme d'une application graphique :
  – `xxgdb` (`http://www.cs.uni.edu/Help/xxgdb.html`)
  – `ddd` (`http://www.gnu.org/software/ddd/`)

A noter également l'existence de `valgrind` qui permet de détecter les fuites de mémoire[6]

1.2 Les mots-clés

Un certains nombres de mots sont réservés pour le langage C. Avant de commencer, il convient donc d'en donner la liste exhaustive :

[4] `man gcc` si vous avez du courage
[5] `http://www.gnu.org/software/binutils/manual/gprof-2.9.1/`
[6] Voir `http://developer.kde.org/~sewardj/docs-2.1.2/manual.html`

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

1.3 Les commentaires

Les commentaires sont très important dans n'importe quel langage de programmation car ils permettent de documenter les fichiers de sources.

```
/* Ceci est un commentaire
   qui peut s'étaler sur plusieurs lignes */

// Ceci est un commentaire qui ne peut s'étaler que sur une ligne
```

Attention car le commentaire sur une ligne est une fonctionnalité qui n'a été définie qu'avec la norme C99 !
Une bonne habitude à prendre dans le domaine général de la programmation est de découper chaque fichier source selon plusieurs niveaux de commentaires :

1. le fichier : pour indiquer le nom de l'auteur, du fichier, les droits de copyright, la date de création, les dates et auteurs des modifications ainsi que la raison d'être du fichier ;
2. la procédure : pour indiquer les paramètres et la raison d'être de la procédure
3. groupe d'instruction : pour exprimer ce que réalise une fraction significative d'une procédure.
4. déclaration ou instruction : le plus bas niveau de commentaire.

On considérera ainsi l'exemple fourni en annexe A.1 page 106.
Attention : Une erreur classique est d'oublier la séquence fermante `*/`.
Dans l'exemple suivantes, les instructions 1 à n seront ignorées à la compilation :
`/* premier commentaire`
Instruction 1
...
Instruction n
`/* deuxième commentaire */`
Il convient également de prendre la bonne habitude de commenté son code source "à la mode Doxygen"[7]. Doxygen est un système de documentation pour de nombreux langages de programmation (C++, C, Java, Objective-C, Python, IDL etc...). Il permet de générer facilement une documentation complète sur un code source en utilisant un format de commentaires particulier.

[7] `www.doxygen.org`

1.4 Structure générale d'un programme C

De manière générale, un programme C consiste en la construction de blocs individuels appelées *fonctions* qui peuvent s'invoquer l'un l'autre. Chaque fonction réalise une certaine tâche[8].
Pour pouvoir s'exécuter, un programme C doit contenir une fonction spéciale appelée `main` qui sera le point d'entrée de l'exécution (c'est à dire la première fonction invoquée au démarrage de l'exécution). Toutes les autres fonctions sont en fait des sous-routines.
Un programme C comporte également la déclaration de *variables* qui correspondent à des emplacements en mémoire. Ces emplacements permettent de stocker des valeurs qui peuvent être utilisées/modifiées par une ou plusieurs fonctions. Les noms des variables sont des identificateurs quelconques (voir §1.5).
Voici deux exemples significatifs :

1. D'abord, le petit classique qui affiche à l'écran "Hello World !" :

```
#include <stdio.h>  // Directive de preprocesseur
void main() {
    printf("Hello world!");
}
```

2. Un exemple un peu plus complet, faisant apparaître toutes les composantes d'un programme C :

```
#include <stdio.h>  // Directive de preprocesseur
#define TVA 15      // idem - la TVA au Luxembourg

float prix_TTC; //déclaration d'une variable externe

/* Exemple de déclaration d'une fonction secondaire */
/* ajoute la TVA au prix HT; renvoie le prix TTC    */
float ajoute_TVA(float prix_HT) {
    return prix_HT*(1 + (TVA/100));
}

/* Fonction main: point d'entree de l'exécution*/
void main() {
    float HT; //déclaration d'une variable interne
    puts("Entrer le prix H.T. : "); // appel de fonctions
    scanf("%f",&HT);                // définies dans stdio.h
    prix_TTC = ajoute_TVA(HT); //appel de notre fonction
    printf("prix T.T.C. : %.2f\n",prix_TTC);
}
```

On voit sur ces deux exemples les différentes composantes d'un programme C :

1. les **directives du préprocesseur** : elles permettent d'effectuer des manipulations sur le texte du programme source, avant la compilation. Une

[8] A noter qu'il existe en C des fonctions précompilées qui peuvent être incluses dans le programme, ces fonctions étant contenues dans des fichiers spéciaux appelés *library files* (extension `*.lib`). Pour accéder à ces fonctions, une directive doit être issue au compilateur indiquant d'inclure les *header files* (extension `*.h`) contenant les déclarations correspondantes à ces fonctions. Dans tous les cas, les autres fonctions doivent être écrites par le programmeur.

directive du préprocesseur est une ligne de programme source commençant par le caractère dièse (`#`). Ces instructions seront détaillées dans le chapitre 7 mais on distingue déjà les deux directives les plus utilisées :
- `#include` qui permet d'inclure un fichier. Ici, le fichier `stdio.h` définit (on préfère dire déclare) les fonctions standards d'entrées/sorties (en anglais STanDard In/Out), qui feront le lien entre le programme et la console (clavier/écran). Dans cet exemple, on utilise les fonctions `puts`, `printf` et `scanf` (voir §2.5).
- `#define` qui définit une constante. Ici, a chaque fois que le compilateur rencontrera le mot TVA, il le remplacera par 15.

2. les **déclarations de variables** : Ces déclarations sont de la forme :

   `type` *nom_variable* `[= <valeur>];`

   Les variables peuvent être déclarées soit de façon *externe* (c'est à dire en dehors d'une fonction), soit de façon *interne* (on dit aussi *locale*) à une fonction.

   Par exemple, dans la déclaration `float prix_TTC;` on a défini la variable externe identifiée par `prix_TTC`, de type `float` (le type des nombres réels dit à virgule flottante, d'où ce nom). Les trois types scalaires de base du C sont l'entier (`int`), le réel (`float`) et le caractère (`char`) et seront abordés au §1.7.

   Toute variable C est typée. Cela permet notamment de savoir quelle place la variable occupera en mémoire et comment il faudra interpréter la suite de bits stockée en mémoire. On peut bien sûr définir ses propres types (voir chapitre 4), mais il existe un certain nombre de types disponibles de façon native. Ces types de base sont détaillés dans la section 1.7.

   Enfin, **on ne peut jamais utiliser de variable sans l'avoir déclarée**.

3. la **définition de fonctions**. Ce sont des sous-programmes dont les instructions vont définir un traitement sur des variables. La déclaration d'une fonction est de la forme :

   `type_resultat` *nom_fonction* `(`$type_1\ arg_1 \ldots type_n\ arg_n$`) {`
   <déclaration de variables locales>
   <liste_d'instructions>
   `}`

   En C, une fonction est définie par :

   (a) une ligne déclarative qui contient :
   - `type_resultat` : le type du résultat de la fonction.
   - *nom_fonction* : le nom qui identifie la fonction.
   - $type_1\ arg_1 \ldots type_n\ arg_n$: les types et les noms des paramètres de la fonction

   (b) un bloc d'instructions délimité par des accolades `{ ... }`, contenant :

– *<déclaration de variables locales >* : les déclarations des données locales (c'est à dire des données qui sont uniquement connues à l'intérieur de la fonction)
– *<liste_d'instructions >* : la liste des instructions qui définit l'action qui doit être exécutée.
Une instructions est une expression terminée par un " `;`". C'est un ordre élémentaire que l'on donne à la machine et qui manipulera les données (variables) du programme. Dans notre exemple, on a vu deux types de manipulation : l'appel de fonctions d'une part (`puts`, `printf`, `scanf` ou `ajoute_TVA`) et une affectation (`=`).
  – `puts` affiche à l'écran le texte fourni en argument.
  – `scanf` attend que l'on entre une valeur au clavier, puis la met dans la variable `HT`, sous format réel (`%f`).
  – `printf` affiche à l'écran un texte formatté.
  Ces fonctions seront détaillées au §2.5 page 27.
Par définition, toute fonction en C fournit un résultat dont le type doit être défini. Le retour du résultat se fait en général à la fin de la fonction par l'instruction `return`. Le type d'une fonction qui ne fournit pas de résultat est déclaré comme `void` (voir §1.7.4).

4. Des **commentaires** (voir §1.3) : ils sont éliminés par le préprocesseur.
A noter que pour ignorer une partie de programme il est préférable d'utiliser une directive du préprocesseur (`#ifdef ... #endif` : voir le chapitre 7).

D'autres exemples de fichiers source sont fournis en annexe A page 106.

1.5 Notion d'identificateur

Un identificateur, comme son nom l'indique, permet de donner un nom à une entité du programme (qu'il s'agisse d'une variable ou d'une fonction). Ils sont sujets aux règles suivantes :

1. Ils sont formés d'une suite de lettres ('a' à 'z' et 'A' à 'Z'), de chiffres (0 à 9) et du signe '_'. En particulier, les lettres accentuées sont interdites ;
2. le premier caractère de cette suite ne peut pas être un chiffre ;
3. les identificateurs sont case-sensitive.

Ainsi, les noms `var1`, `S_i`, `_start` et `InitDB` sont des identificateurs valides, tandis que `i:j` ou `1i` ne le sont pas.

1.6 Conventions d'écritures d'un programme C

Avant d'aller plus loin, il convient d'établir un certain nombre de règles de présentation que devra suivre tout bon programmeur qui souhaite écrire des programmes C lisibles[9] :
– ne jamais placer plusieurs instructions sur une même ligne ;

[9]Ces règles peuvent largement être étendues à tout langage de programmation

– utiliser des identificateurs significatifs ;
– grâce à l'indentation des lignes, faire ressortir la structure syntaxique du programme ;
– laisser une ligne blanche entre la dernière ligne des déclarations et la première ligne des instructions ;
– aérer les lignes de programme en entourant par exemple les opérateurs avec des espaces ;
– bien commenter les listings tout en évitant les commentaires triviaux.

A ce propos, la lecture de [Her01] devrait permettre au lecteur d'aquérir de bon réflexes de programmation.

1.7 Les types de base

1.7.1 Les caractères

On utilise le mot-clé `char` pour désigner une variable de type char. Il s'agit en fait d'un entier codé sur 8 bits interprété comme un caractère utilisé sur la machine (il s'agit en général du code ASCII de ce caractère).
Ex :

```
char c1 = 'a'; // Déclaration d'une variable c1 de type char
               // a laquelle on affecte la valeur 'a'
               // A noter l'utilisation du simple quote
char c2 = 97;  //c2 correspond également au caractère 'a'
```

Le tableau 1.1 donne la liste des principaux codes ASCII en décimal (pour être précis, il s'agit de la première moitié des caractère ASCII, l'autre moitié correspondants en général à des caractères étendus comme Ä).

code	0	1	2	3	4	5	6	7	8	9
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT
10	LF	VT	NP	CR	SO	SI	DLE	DC1	DC2	DC3
20	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS
30	RS	US	SP	!	”	#	$	%	&	’
40	(	)	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[	\	]	^	_	‘	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{	|	}	-	DEL		

Tab. 1.1 – Codes ASCII en décimal

Caractères particuliers

Il existe un certain nombre de caractères particuliers dont les principaux sont résumés dand le tableau 1.2.

Caractère	`'\n'`	`'\t'`	`'\b'`	`'\''`	`'\"'`	`\?`
Sémantique	*retour à la ligne*	*tabulation*	*backspace*	*'*	*"*	*?*

TAB. 1.2 – Quelques caractères spéciaux

Ces caractères ont une signification particulière pour le langage C. Par exemple, le caractère `"` **doit** être utilisé dans les chaînes de caractère (et la séquence `'\"'` permet de représenter ce caractère dans une chaîne). De même, le caractère `?` est utilisé dans les *trigraphes* qui sont des séquences de trois caractères permettant de désigner les caractères `# [ ] \ ^ { } | ~`. Le tableau 1.3 détaille les trigraphes disponibles.

Trigraphe	`??=`	`??(`	`??/`	`??)`	`??'`	`??<`	`??!`	`??>`	`??-`
Signification	`#`	`[`	`\`	`]`	`^`	`{`	`|`	`}`	`~`

TAB. 1.3 – Les 9 trigraphes disponibles en norme ANSI

Et les chaînes de caractères ?

En empiétant un peu sur la suite, les chaînes de caractères sont vues comme un pointeur sur des caractères et sont donc de type `char *`. Ex :

```
char * chaine = "Hello World !";// une chaine de caractère
                                // noter l'utilisation du double
                                // quote
```

Plus de détails dans le chapitre 3 consacré aux pointeurs.

1.7.2 Les entiers

On utilise alors le mot-clé `int`. Exemple :

```
/* déclaration la plus courante d'une variable de type int */
int a = 14; // la variable a est initialisée à la valeur 14

/* Utilisation des précisions (cas le + général)*/
short int b; // b est codé sur 16 bits
int c;       // c est codé sur 16 ou 32 bits
long int d;  // d est codé sur 32 bits

// la possibilité de l'écriture suivante dépend du compilateur
long long int e; // d est codé sur 64 bits.
```

```
/* Précision du signe */
unsigned long int n; //n est un entier non signé sur 32 bits
```

Par défaut, les entiers sont en représentation signée (principalement en représentation par complément à 2 pour les acharnés).
Ainsi, un `short int` peut contenir tout entier donc la valeur est comprise entre -2^{15} et $2^{15}-1$. Au contraire, un `unsigned short int` (non signé) pourra contenir un entier compris entre 0 et $2^{16}-1$.
Le tableau 1.4 regroupe les types entiers standards avec quelques informations supplémentaires (la taille utilisée en mémoire et l'intervalle des valeurs possibles. Voici quelques valeurs numériques pour avoir un meilleur ordre d'idée des intervalles utilisés :

$2^{15} = 32.768$ $2^{16} = 65.536$
$2^{31} = 2.147.483.648$ 2^{32}=4.294.967.295 .
2^{63}=9.223.372.036.854.775.808 2^{64}=18.446.744.073.709.551.616

Type	**Taille mémoire**	**Intervalle de valeurs**
char	1 octet	[-128 ;127] ou [0,255]
unsigned char	1 octet	[0 ;255]
signed char	1 octet	[-128 ;127]
int	2 ou 4 octets	$[-2^{15};2^{15}-1]$ ou $[-2^{31};2^{31}-1]$
unsigned int	2 ou 4 octets	$[0;2^{16}-1]$ ou $[0;2^{32}-1]$
short int	2 octets	$[-2^{15};2^{15}-1]$
unsigned short int	2 octets	$[0;2^{16}-1]$
long	4 octets	$[-2^{31};2^{31}-1]$
unsigned long	4 octets	$[0;2^{32}-1]$
long long(*)	8 octets	$[-2^{63};2^{63}-1]$
unsigned long long(*)	8 octets	$[0;2^{64}-1]$

TAB. 1.4 – Les types entiers standards. (*) : dépend du compilateur utilisé

Remarque : les valeurs limites des differents types sont indiquées dans le fichier header `<limits.h>` (voir §9.9). En principe, on a
`sizeof(short) <= sizeof(int) <= sizeof(long)`

Cas des constantes entières

On distingue 3 notations :
– décimale (écriture en base 10) : c'est l'écriture usuelle. Ex : `372` ;
– octale (base 8) : on commence par un `0` suivi de chiffres octaux. Ex : `0477` ;
– héxadécimale (base 16) : on commence par `0x` (ou `0X`) suivis de chiffres héxadécimaux (0-9 a-f). Ex : `0x5a2b`, `0X5a2b`, `0x5A2b`.

Par défaut, le type des constantes entières est le type `int`. On peut aussi spécifier explicitement le type en ajoutant l'un des suffixes `L` ou `l` (pour le type `long`), `LL` or `ll` (pour le type `long long`). On peut aussi ajouter le suffixe `U` ou `u` (pour `unsigned`). Des exemples sont fournis dans le tableau suivant :

Décimal	Octal	Hexa	Type
15	017	Oxf	int
32767	077777	0x7FFF	int
10U	012U	0xAU	unsigned int
32768U	0100000U	0x8000u	unsigned int
16L	020L	0x10L	long
27UL	033ul	0x1BUL	unsigned long

1.7.3 Les flottants

On distingue trois types de flottants allant de la précision la plus faible à la plus forte (qui dépend de l'implémentation) : `float`, `double` et `long double`. Ex : `double Pi = 3.14159;`

Representation Interne

La taille de stockage et la représentation interne des nombres flottant dépend du compilateur utilisé mais la plupart utilise le standard IEEE 754-1985[Ste90]. Un nombre flottant x est ainsi composé d'un signe s ($s = 1$ pour un nombre négatif, 0 sinon) , d'une mantisse m et un exposant exp en base 2 tels que :

$$x = s * m * 2^{exp} \text{ avec } \begin{cases} 1.0 \leq m < 2 \textbf{ ou} \\ m = 0 \end{cases}$$

La *précision* du nombre flottant détermine le nombre de bits utilisée pour la mantisse, alors l'*intervalle de valeur* est déterminé par le nombre de bits utilisés par l'exposant. Le tableau 1.5 détaille ces informations pour chaque type flottant.

Type	Taille mémoire	Intervalle de valeurs	Précision
float	4 octets	$[1,2 * 10^{-38}; 3,4 * 10^{38}]$	6 chiffres décimaux
double	8 octets	$[2,3 * 10^{-308}; 1,7 * 10^{308}]$	15 chiffres décimaux
long double	10 octets	$[3,4 * 10^{-4932}; 1,1 * 10^{4932}]$	19 chiffres décimaux

TAB. 1.5 – Les types flottant pour les nombres réels.

La figure 1.2 montre le format de stockage d'un nombre de type `float` dans la représentation IEEE :

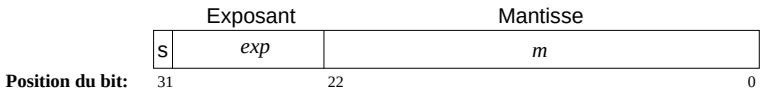


FIG. 1.2 – Format de stockage IEEE pour un nombre de type float

En binaire, le premier bit de la mantisse non nulle sera toujours 1 : il n'est donc pas représenté. L'exposant est stocké avec un biais (qui vaut 127 pour le type `float`) Un petit exemple pour bien comprendre : $-2,5 = -1 * 1,25 * 2^1$. Les valeurs stockées seront donc : $S = 1$; $exp = 127 + 1$; $m = 0.25$

Cas des constantes flottantes

Des exemples d'utilisation des constantes flottantes sont résumées dans le tableau suivant :

notation C	correspondance	notation C	correspondance
`2.`	2	`.3`	0.3
`1.4`	1.4	`2e4`	$2 * 10^4$
`2.e4`	$2 * 10^4$	`.3e4`	$0.3 * 10^4$
`1.4e-4`	$1.4 * 10^{-4}$		

1.7.4 Le type void

On a vu que toute variable C était typée, de même que toute valeur de retour d'un fonction. Mais il peut arriver qu'aucune valeur ne soit disponible. Pour exprimer l'idée de "aucune valeur", on utilise le mot-clé `void`. Ce type est utilisé dans trois situations :

1. les expressions de type `void` : on les rencontre principalement dans la déclaration de fonctions qui n'ont pas de valeur de retour.
   Ex : `void exit (int status);`
   On utilise aussi dans le cadre d'une conversion de type (on parle de *cast*, voir §2.4) vers le type `void`, ce qui n'a de sens que si la valeur résultat n'est pas utilisée. Ex : `(void)printf("Un exemple.");`
2. le prototype de fonctions sans paramètres : `int rand(void);`
   A noter qu'en pratique, on écrira plus simplement : `int rand();`
3. les pointeurs vers le type `void` : le type `void *` est le type pointeur générique, c'est à dire qu'il est capable de pointer vers n'importe quel type d'objet. Il représente donc l'adresse de l'objet et non son type. Plus de précisions dans la section3 page 33 consacrée aux pointeurs.

Chapitre 2

La syntaxe du langage

2.1 Expressions et Opérateurs

Une expression est une combination d'opérateurs et d'opérandes. Dans les cas les plus simples, une expression se résume à une constante, une variable ou un appel de fonction. Des expressions peuvent également être utilisées comme opérandes et être jointes ensembles par des opérateurs pour obtenir des expressions plus complexes.
Toute expression a un type et si ce type **n'est pas** `void`, une valeur. Quelques exemples d'expressions :

```
4 * 512           // Type: int
4 + 6 * 512       // Type: int; equivalent to 4 + (6 * 512)
printf("Un exemple!\n") // Appel de fonction, type: int
1.0 + sin(x)            // Type: double
srand((unsigned)time(NULL))     // Appel de fonction; type: void
(int*)malloc(count*sizeof(int)) // Appel de fonction; type: int *
```

Les opérateurs peuvent être *unaires* (à une opérande) ou *binaires* (à deux opérandes). Il existe un seul opérateur *ternaire* (il s'agit de `?:`) Les paragraphes qui suivent détaillent les opérateurs disponibles.

2.1.1 Opérateurs arithmétiques

Les principaux opérateurs arithmétiques sont résumés dans le tableau 2.1. Les opérandes de ces opérateurs peuvent appartenir à tout type arithmétique[1]
Quelques remarques :
- On peut effectuer une *conversion de type* aux opérandes. Le résultat de l'opération prend le type de cette conversion. Ainsi ; 2.0/3 est équivalent à 2.0/3.0 et le résultat est de type `float`.
- Le résultat d'une division d'entiers est aussi un entier ! Ex :
  ```
  6 / 4       // Resultat:  1
  6 % 4       // Resultat:  2
  6.0 / 4.0   // Resultat: 1.5
  ```

[1] Seul l'opérateur % requiert des types entiers

Opérateur	Traduction	Exemple	Résultat
`+`	Addition	`x + y`	l'addition de x et y
`-`	Soustraction	`x - y`	la soustraction de x et y
`*`	Produit	`x * y`	la multiplication de x et y
`/`	Division	`x / y`	le quotient de x et y
`%`	Reste	`x % y`	Reste de la division euclidienne de x par y
`+`(unaire)	Signe positif	`+x`	la valeur de x
`-`(unaire)	Signe négatif	`-x`	la négation arithmétique de x
`++`(unaire)	Incrément	`x++` ou `++x`	x est incrémenté ($x = x + 1$). L'opérateur préfixe `++x` (resp. suffixe `x++`) incrémente x avant (resp. après) de l'évaluer
`--`(unaire)	Decrément	`x--` ou `--x`	x est décrementé ($x = x - 1$). L'opérateur préfixe `--x` (resp. suffixe `x--`) décrémente x avant (resp. après) de l'évaluer

TAB. 2.1 – Les principaux opérateurs arithmétiques

– Concernant l'incrémentation pré/postfixe, voici un petit exemple pour bien comprendre :
  Supposons que la valeur de N soit égale à 5 :
  – Incrémentation postfixe : `X = N++`; Résultat : $N = 6$ et $X = 5$
  – Incrémentation préfixe : `X = ++N`; Résultat : $N = 6$ et $X = 6$

2.1.2 Opérateurs d'affectation

Opérateur	Traduction	Exemple	Résultat
`=`	affectation simple	`x = y`	assigne la valeur de y à x
`(op)=`	affectation composée	`x += y`	`x (op)=y` est équivalent à `x = x (op) y`

TAB. 2.2 – Les opérateurs d'affectation

On distingue deux types d'opérateurs d'assignement qui sont résumés dans le tableau 2.2. L'opérande de gauche doit être une expression qui désigne un objet (une variable par exemple).
Dans l'écriture `x(op)=y`, la variable x n'est évaluée qu'une seule fois. (op) est un opérateur arithmétique ou de manipulation de bits Les opérateurs d'affectation composés sont donc :

`+=` `-=` `*=` `/=` `%=` `&=` `^=` `|=` `<<=` `>>=`

2.1.3 Opérateurs relationnels

Toute comparaison est une expression de type `int` qui renvoie la valeur 0 (false) ou 1 (true). Il faut que les opérandes soient du même type arithmétique (ou des pointeurs sur des objets de même type).

Opérateur	Traduction	Exemple	Résultat
`<`	inférieur	`x < y`	1 si x est inférieur à y
`<=`	inférieur ou égal	`x <= y`	1 si x est inférieur ou égal à y
`>`	supérieur	`x > y`	1 si x est supérieur à y
`>=`	supérieur ou égal	`x >= y`	1 si x est supérieur ou égal à y
`==`	égalité	`x == y`	1 si x est égal à y
`!=`	inégalité	`x != y`	1 si x est différent de y

TAB. 2.3 – Les opérateurs relationnels

Les différents opérateurs relationnels sont détaillés dans le tableau 2.3.
Attention : Une erreur **ultra classique** est de confondre l'opérateur d'égalité (`==`) avec celui d'affectation (`=`). Ainsi, considérons le code suivant[2] :

```
/* Fichier: erreur.c */
#include <stdio.h>
int main () {
    int x=14,y=1; // x est différent de y
    if (x = y)    //erreur!!! il faudrait écrire 'if (x == y)'
            printf("x est égal à y (%i=%i)\n",x,y);
    else
            printf("x est différent de y (%i!=%i)\n",x,y);
    return 0;
}
```

A priori, x et y sont différents et on s'attend à obtenir un message le confirmant. Néanmoins, l'erreur d'ecriture a permis d'effectuer une affectation de la valeur de x si bien que ce programme renvoit la ligne[3] : `x est égal à y (1=1))` (noté la valeurs de x)
Avec une écriture correcte, on aurait obtenu : `x est différent de y (14!=1)`
Remarque : en initialisant `y` a 0 (et non a 1), on aurait obtenu le message `x est différent de y (0 !=0)` car le résultat de l'affectation est la valeur affectée (ici 0). Cette valeur est considérée comme 'fausse' pour la condition testée dans le `if` si bien que les instruction de la branche `else` sont exécutées (voir §2.2.1).

2.1.4 Opérateurs logiques

Les opérateurs logiques, listés dans le tableau 2.4 permettent de combiner le résultat de plusieurs expressions de comparaison en une seule expression logique. Les opérandes des opérateurs logiques peuvent être n'importe quel scalaire (i.e arithmétique ou pointeur). Toute valeur différente de 0 est interprétée comme vraie (et 0 correspond à 'faux'). Comme pour les expressions relationnelles, les expressions logiques renvoient une valeur entière (0=false ; 1=true).
Remarque : les opérateurs `&&` et `||` évaluent les opérandes de gauche à droite et le résultat est connu dès l'opérande de gauche. Ainsi, l'opérande de droite n'est

[2]Compiler ce code avec la commande `gcc -Wall erreur.c -o erreur`
[3]l'option de compilation '-Wall' a quand même soulevé un warning

Opérateur	Traduction	Exemple	Résultat
`&&`	ET logique	`x && y`	1 si x et y sont différents de 0
`||`	OU logique	`x || y`	1 si x et/ou y sont différents de 0
`!`	NON logique	`!x`	1 si x est égal à 0. Dans tous les autres cas, 0 est renvoyé.

TAB. 2.4 – Les opérateurs logiques

évaluée que si celle de gauche est vraie dans le cas de l'opérateur `&&` (respectivement fausse dans le cas de l'opérateur `||`). Par exemple, dans l'expression `(i < max) && (f(14) == 1)`, la fonction `f` n'est appelée que si $i < max$.

2.1.5 Opérateurs bit à bit

Opérateur	Traduction	Exemple	Résultat (pour chaque position de bit)
`&`	ET bit à bit	`x & y`	1 si les bits de x et y valent 1
`|`	OU bit à bit	`x | y`	1 si le bit de x et/ou de y vaut 1
`^`	XOR bit à bit	`x ^ y`	1 si le bit de x ou de y vaut 1
`~`	NON bit à bit	`~x`	1 si le bit de x est 0
`<<`	décalage à gauche	`x << y`	décale chaque bit de x de y positions vers la gauche
`>>`	sécalage à droite	`x >> y`	décale chaque bit de x de y positions vers la droite

TAB. 2.5 – Les opérateurs de manipulation des bits

Les opérateurs bits à bits n'opèrent que sur des entiers. Les opérandes sont interprétées bits par bits (le bit 1 correspondant à une valeur vraie, 0 est considéré comme une valeur fausse). Quelques exemples pour bien comprendre (chaque opérande est fournie sous forme décimale et binaire) :

x		y		Opération	Résultat	
14	1110	9	1001	`x & y`	8	1000
14	1110	9	1001	`x | y`	15	1111
14	1110	9	1001	`x ^ y`	7	0111
14	1110			`~x`	1	0001
14	1110	2	0010	`x << y`	56	111000
14	1110	2	0010	`x >> y`	3	11

2.1.6 Opérateurs d'accès à la mémoire

L'opérande de l'opérateur d'adresse `&` doit être une expression qui désigne un objet ou une expression. Ainsi, `&x` renvoie l'adresse mémoire de x et est donc un pointeur vers x. Plus de détails dans le chapitre 3 dédié aux pointeurs.

Op.	Traduction	Exemple	Résultat
`&`	Adresse de	`&x`	l'adresse mémoire de x
`*`	Indirection	`*p`	l'objet (ou la fonction) pointée par p
`[ ]`	Elément de tableau	`t[i]`	L'équivalent de `*(x+i)`, l'élément d'indice i dans le tableau t
`.`	Membre d'une structure ou d'une union	`s.x`	le membre x dans la structure ou l'union s
`->`	Membre d'une structure ou d'une union	`p->x`	le membre x dans la structure ou l'union pointée par p

TAB. 2.6 – Les opérateurs d'accès à la mémoire

Les opérateurs d'accès aux membres d'une structure ou d'une union seront plus amplement détaillés dans les §4.3 et §4.4.

2.1.7 Autres opérateurs

Il existe un certain nombre d'autres opérateurs qui sont listés dans le tableau 2.7. On y retrouve notamment l'appel de fonction et la conversion de type (qui ne s'applique qu'aux types scalaires et qui sera abordée plus en détail dans la section 2.4 page 25).

Op.	Traduction	Exemple	Résultat
`()`	Appel de fonction	`f(x,y)`	Exécute la fonction f avec les arguments x et y
`(type)`	cast	`(long)x`	la valeur de x avec le type spécifié
`sizeof`	taille en bits	`sizeof(x)`	nombre de bits occupé par x
`? :`	Evaluation conditionnelle	`x?:y:z`	si x est différent de 0, alors y sinon z
`,`	séquencement	`x,y`	Evalue x puis y

TAB. 2.7 – Les autres opérateurs

L'opérateur `sizeof` renvoie le nombre de bits requis pour stocker un objet du type spécifié. Le résultat est une constante de type `size_t`.
L'opérateur `?:` permet une ecriture plus compacte de l'évaluation conditionnelle `if...then...else`. Ainsi l'expression :

```
(x >= 0) ? x : -x
```

renvoie la valeur absolue de de x. Plus de détails au §2.2.1.

2.2 Les structures de contrôle

Comme pour la plupart des langages de programmation, le langage C définit un certain nombre de structures de contrôle (boucles et branchements).

2.2.1 Instruction if...else

Syntaxe :

1. `if` (*expression*) *Instruction1*
2. `if` (*expression*) *Instruction1* `else` *Instruction2*

La valeur de *expression* est évaluée et si elle est différente de 0, *Instruction1* est exécutée, sinon *Instruction2* est exécutée (si elle existe). Exemple :

```
if (x > y)  max = x; // Assigne la plus grande valeur entre x et y
else        max = y; // à la variable max.
```

Remarque :
– l'exemple précédent aurait pu s'écrire plus simplement (mais moins lisiblement) : `(x > y)?(max = x):(max = y);` ou même `max = (x>y)? x : y;`
– Attention aussi au fait que *expression* doit être correctement parenthésée.
– la partie 'then' n'est pas introduite par le mot-clé `then` comme dans certains langages.

2.2.2 Instruction for

Syntaxe :
– `for` (*expression*$_1$; *expression*$_2$; *expression*$_3$)
Instruction
expression$_1$ et *expression*$_3$ peuvent être n'importe quelle expression. *expression*$_2$ est une expression de contrôle et doit donc être de type scalaire. Le déroulement de cette instruction est illustré par l'organigramme de la figure 2.1.

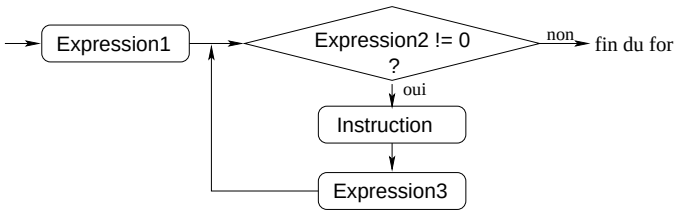


FIG. 2.1 – Organigramme de l'instruction FOR

Un exemple pour bien comprendre :

```
int i,MAX=14; //compteur
for (i=0; i < MAX ; i++) {
   printf("Valeur de i : %i\n",i);
}
```

2.2.3 Instruction while

Syntaxe :
– `while` (*expression*) *Instruction*
Il s'agit d'une boucle : tant que *expression* est vraie, *Instruction* est exécutée, conformément à l'organigramme de la figure 2.2.
Exemple :

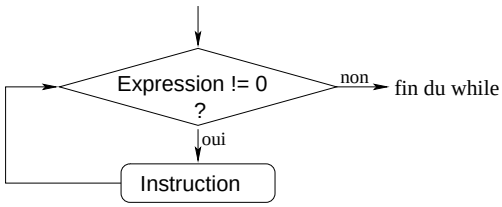


FIG. 2.2 – Organigramme de l'instruction WHILE

```
#define MAX 14
int i=0;
while (i < MAX ) {
   printf("Valeur de i : %i\n",i);
   i++;
}
```

2.2.4 Instruction do...while

Syntaxe :

`do` *instruction* `while` (*expression*) ;

l'instruction `do...while` a un fonctionnement globalement similaire à celui d'une boucle `while` mise à part le fait que le corps de la boucle est exécuté avant que l'expression de contrôle soit évaluée, comme dans l'organigramme de la figure 2.3. Ainsi, le corps de la boucle est **toujours** exécuté au moins une fois.

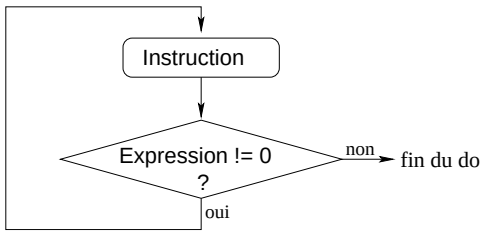


FIG. 2.3 – Organigramme de l'instruction DO

L'exemple suivant imprimera "`Valeur de i : 14`" (alors que $i \geq 14$) :

```
#include <stdio.h>
#define MAX 14
int main() {
    int i=MAX;
    do {
        printf("Valeur de i : %i\n",i);
        i++;
    } while (i < MAX);
    return 0;
}
```

2.2.5 Instruction switch

Cette instruction est un `if` généralisé. Sa syntaxe est la suivante :
`switch` (*expression*) {
`case` $constante_1$: $liste\_d'instructions_1$ `break`;
...
`case` $constante_n$: $liste\_d'instructions_n$ `break`;
`default` : *liste_d'instructions*
}

1. *expression* est évaluée, puis le résulat est comparé avec $constante_1$, $constante_2$ etc...
2. A la première $constante_i$ dont la valeur est égale à *expression*, la (ou les[4]) *liste_d'instructions* correspondante(s) est exécutée jusqu'à la rencontre d'une instruction `break`. La rencontre d'un `break` termine l'instruction `swith`.
3. s'il n'existe aucune $constante_i$ dont la valeur soit égale à celle de *expression*, on exécute la *liste_d'instructions* de l'alternative `default` si celle-ci existe, sinon rien n'est fait.

ATTENTION : *expression* est nécessairement une valeur entière (voir le tableau 1.4 page 12). C'est pourquoi on utilisera souvent dans les exemples qui suivent des enumérations (voir §4.1 page 41).
Compte tenu du nombre d'éléments optionnels, on peut distinguer 3 types d'utilisations possibles :

1. pour chaque alternative `case`, on trouve un `break`. Exemple :

```
enum {FALSE, TRUE};
void print_boolean(int bool) {
   switch (bool) {
      case FALSE: printf("faux"); break;
      case TRUE:  printf("vrai"); break;
      default: printf("Erreur interne");
   }
}
```

2. on peut avoir une ou plusieurs alternatives ne possédant ni *liste_d'instructions*, ni `break`. Par exemple, si on veut compter le nombre de caractères qui sont des chiffres et ceux qui ne le sont pas, on peut utiliser le `switch` suivant :

```
switch (c) {
   case '0':
   case '1':
   case '2':
   case '3':
   case '4':
   case '5':
   case '6':
   case '7':
   case '8':
```

[4] l'instruction `break` est optionnelle

```
      case '9': nb_chiffres++; break;
      default:  nb_non_chiffres++;
   }
```

3. enfin on peut ne pas avoir de `break` comme dans l'exemple suivant :

```
enum {POSSIBLE, IMPOSSIBLE};
void print_cas(int cas) {
   switch (cas) {
      case IMPOSSIBLE: printf("im");
      case POSSIBLE: printf("possible");
   }
}
```

L'instruction `break` peut être étendue à d'autres cas de branchements (ou de sauts) comme on le verra au §2.2.7.
Les paragraphes qui suivent abordent les branchement non conditionnels (par opposition aux branchements conditionnels correspondant aux instructions `if...else` et `switch`) qui permettent de naviguer au sein des fonctions du programme.

2.2.6 Instruction goto

Syntaxe :

`goto` *etiquette* ;

La directive `goto` permet de brancher directement à n'importe quel endroit de la fonction courante identifiée par une *étiquette*. Une étiquette est un identificateur suivi du signe ":".
Exemple :

```
for ( ... )
    for ( ... )
       if ( erreur )
         goto  TRAITEMENT_ERREUR;
  ...
TRAITEMENT_ERREUR:      // le traitement d'erreur est effectué ici
   printf("Erreur: ....");
   ...
```

Remarque :
- Encore une fois, l'instruction `goto` et la déclaration de l'etiquette doivent être contenu au sein de la même fonction.
- N'utiliser cette instruction que lorsque vous ne pouvez pas faire autrement. Le plus souvent, vous pouvez vous en passer alors n'en abusez pas !
- Pour une saut en dehors d'une même fonction, on pourra utiliser les fonctions `setjmp` et `longjmp`.

2.2.7 Instruction break

Syntaxe :

`break;`

On a vu le rôle de l'instruction `break` au sein de la directive de branchement multiple `switch`. On peut l'utiliser plus généralement au sein de n'importe

quelle boucle (instructions `for`, `while` ou `do...while`) pour interrompre son déroulement et passer directement à la première instruction qui suit la boucle. Exemple :

```
while (1) {
...
   if (command == ESC) break;  // Sortie de la boucle
...
}
// on reprend ici après le break
```

En cas de boucles imbriquées, `break` fait sortir de la boucle la plus interne

2.2.8 Instruction continue

Syntaxe :

```
   continue;
```

L'instruction `continue` ne peut être utilisée que dans le corps d'une boucle (instruction `for`, `while` ou `do`). Elle permet de passer directement à l'itération suivante.
Exemple :

```
for (i = -10; i < 10; i++) {
  if (i == 0) continue; // passer à 1 sans exécuter la suite
...
}
```

2.3 La récursivité

Comme pour beaucoup de langages, la récursivité est possible en C. Qu'est-ce que la récursivité ? C'est la propriété qu'a une fonction de s'auto-appeler, c'est-à-dire de se rappeler elle-même plusieurs fois. La notion de récurrence est largement présente dans le domaine mathématique, notamment dans la définition des suites. Exemple :

$$\begin{cases} u_0 = 1 \\ u_n = 2u_{n-1} + 1 \quad \forall\ n \geq 1 \end{cases}$$

Les premiers termes consécutifs de cette suite sont donc :
$u_0 = 1$
$u_1 = 2u_0 + 1 = 3$
$u_2 = 2u_1 + 1 = 7 \ldots$
Bien sûr, on pourrait calculer explicitement la valeur de u_n en fonction de n (ici, on peut montrer que $\forall n \in \mathbb{N}, u_n = 2^{n+1} - 1$) mais on peut utiliser directement la définition de la fonction pour la programmation du calcul correspondant :

```
#include <stdio.h>
/*************************************************************
 * Exemple de fonction traduisant la définition de la suite:
 *   u_0 = 1
 *   u_n = 2*u_{n-1} + 1  si n >= 1
```

```
 * Contrainte: n>=0
 ************************************************************/
int ma_suite(unsigned int n) {
    if (n == 0) return 1; //u_0 = 1
    else return 2*ma_suite(n-1) + 1; //appel récursif
}
/*** Fonction main: point d'entrée de l'exécution ***/
int main() {
    unsigned int n;
    puts("Entrer la valeur de n : ");
    scanf("%u",&n);
    printf("valeur de la suite : %u\n",ma_suite(n));
    return 0;
}
```

2.4 Les conversions de types

Une *conversion de type* renvoie la valeur d'une expression (de type A) dans un nouveau type B. Les conversions ne peuvent être effectuées que sur les types scalaires (c'est à dire les types arithmétiques et les pointeurs).
Une conversion de type conserve toujours la valeur originale dans la mesure ou le nouveau type est capable de la représenter. Ainsi, un nombre flottant peut être arrondi dans une conversion de `double` vers `float`. Enfin, la conversion peut être soit *implicite*, c'est à dire effectuée par le compilateur (par exemple, si i est une variable de type `float` et j de type `int`, alors l'expression `i+j` est automatiquement de type `float`).
Elle peut être également *explicite* grâce à l'opérateur de *cast* (transtypage en français). Il est de bon goût et très important (pour la clarté du programme) d'utiliser l'opérateur de cast dès qu'une conversion de type est requise. Cela évite aussi les warnings du compilateur. Exemple :

```
void f(float i, float j) { // fonction avec 2 paramètres de type
   ...                     // float
}
int main() {
   int i = 14;
   float j = 2.0;
   ...
   f( (float)i, j); // appel de f avec conversion de i vers
                    // le type float
}
```

2.4.1 Les situations de la conversion de type

En C, les situations où se produisent les conversions sont les suivantes :

1. une valeur d'un certain type est utilisée dans un contexte qui en requiert un autre :
   – passage de paramètre : le paramètre effectif n'a pas le type du paramètre formel (c'était le cas dans l'exemple précédent) ;

- affectation : la valeur à affecter n'a pas le même type que la variable ;
- valeur rendue par une fonction : l'opérande de *return* n'a pas le type indiqué dans la déclaration de la fonction.

2. opérateur de conversion : le programmeur demande explicitement une conversion.

3. un opérateur a des opérandes de types différents.

Dans ce dernier cas, et contrairement aux deux cas précédents, c'est le compilateur qui effectue la conversion selon des règles soigneusement définies qui sont détaillées dans les deux paragraphes qui suivent.

2.4.2 La règle de "promotion des entiers"

Cette règle est appliquée aux opérandes des opérateurs unaires `+` et `-`, ainsi qu'aux opérateurs binaires de décalage `<<` et `>>` et dans les *conversions arithmétiques habituelles*, abordées au §2.4.3. Elle a pour but d'amener les "petits entiers" à la taille des `int`.
Ainsi, toute opérande de type `char`, `unsigned char`, `short` , `unsigned short` ou champs de bits sur les opérateurs précédent est automatiquement convertie en `int` (ou en `unsigned int` si le type `int` ne permet pas de représenter l'opérande).

2.4.3 Les conversions arithmétiques habituelles

Cette règle est appliquée à tous les opérateurs arithmétiques binaires, exceptés les opérateurs de décalage `<<` et `>>` et les seconde et troisième opérandes de l'opérateur `?:`.
La règles qui s'applique est la suivante :
Dans le cas d'opérandes entières, on applique déjà la règle de promotion des entiers, ce qui permet de se débarrasser des petits entiers. Ensuite, si les opérandes sont toujours de types différents, on les convertit dans le type le plus haut dans la hiérarchie exposée dans la figure 2.4.

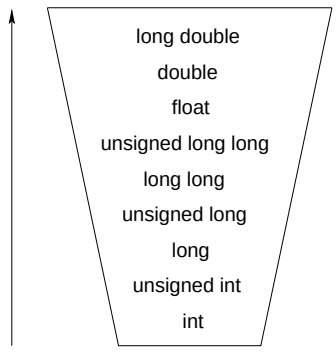


FIG. 2.4 – Hiérarchie des conversions arithmétiques habituelles

2.4.4 Les surprises de la conversion de type

En général, les conversions fonctionnent de façon satisfaisante vis à vis du programmeur. Cependant, il existe une situation permettant d'obtenir des résultats surprenants : lors de la *comparaison entre des entiers signés et non signés.* Considérer ainsi le programme suivant :

```
#include <stdio.h>
int main () {
   unsigned int i = 0;
   if (i < -1) printf("Cha ch'est louche\n");
   else printf("On est super content!\n");
   return 0;
}
```

Celui ci imprimera le message `Cha ch'est louche`, laissant croire que $0 < -1$... L'explication est la suivante : l'opérateur `<` possède une opérande de type `unsigned int` (i) et l'autre de type `int` (la constante -1). D'après la figure 2.4, cette dernière est convertie en `unsigned int` et le compilateur réalise la comparaison non signée entre 0 et 4294967295 (puisque -1 = `0xFFFFFFFF` = 4294967295). CQFD.
Pour que tout revienne dans l'ordre, il suffit d'utiliser l'opérateur de conversion dans le test qui devient : `if ( (int) i < -1)`
Attention car on peut aussi utiliser des `unsigned int` sans s'en rendre compte ! Ainsi :

```
#include <stdio.h>
int main () {
   if ( sizeof(int) < -1) printf("cha ch'est louche\n");
   else  printf("On est super content!\n");
   return 0;
}
```

imprimera le message... `Cha ch'est louche`. Pourtant, les `int` n'ont pas une longueur négative... Il se trouve simplement qui dans la déclaration de la fonction `sizeof` (définie dans `stddef.h`) renvoie un élément de type `size_t`, un entier non signé...
Conclusion :

1. Ne **jamais** mélanger entiers signés et non signés dans les comparaisons (l'option `-Wall` du compilateur le signale normalement) ! Utiliser l'opérateur de conversion pour amener les opérandes de la comparaison dans le même type.
2. bien noter que `sizeof` renvoit une valeur de type entier non signé.

2.5 Principales fonctions d'entrées-sorties standard

Il s'agit des fonctions de la librairie standard `stdio.h` (voir aussi le §9.18 et le chapitre 6) utilisées avec les unités classiques d'entrées-sorties, qui sont respectivement le clavier et l'écran. Sur certains compilateurs, l'appel à la librairie `stdio.h` par la directive au préprocesseur `#include <stdio.h>` n'est pas nécessaire pour utiliser les fonctions présentées ici, notamment `printf` et `scanf`.

2.5.1 La fonction `getchar`

La fonction `getchar` permet la récupération d'un seul caractère à partir du clavier. La syntaxe d'utilisation de `getchar` est la suivante :

```
var=getchar();
```

Notez que `var` doit être de type `char`. Exemple :

```
#include <stdio.h>
int main() {
    char c;
    printf("Entrer un caractère:");
    c = getchar();
    printf("Le caractère entré est %c\n",c);
    return 0;
}
```

A noter enfin que cette fonction est strictement équivalente à `getc(stdin)` (Cette fonction sera abordée au chapitre 6).

2.5.2 La fonction `putchar`

La fonction `putchar` permet l'affichage d'un seul caractère sur l'écran de l'ordinateur. `putchar` constitue alors la fonction complémentaire de `getchar`. La syntaxe d'utilisation est la suivante :

```
putchar(var);
```

où `var` est de type `char`. Exemple :

```
#include <stdio.h>
int main() {
    char c;
    printf("Entrer un caractère:");
    c = getchar();
    putchar(c);
    return 0;
}
```

2.5.3 La fonction `puts`

Syntaxe :

```
puts(ch);
```

Cette fonction affiche, sur `stdout`, la chaîne de caractères `ch` puis positionne le curseur en début de ligne suivante. `puts` retourne EOF en cas d'erreur. Exemple :

```
#include <stdio.h>
int main() {
    char * toto = "on est super content!";
    puts(toto);
    return 0;
}
```

2.5.4 La fonction d'écriture à l'écran formattée `printf`

La fonction `printf` est une fonction d'impression formatée, ce qui signifie que les données sont converties selon le format particulier choisi. Sa syntaxe est la suivante :

`printf("chaîne de contrôle",` $expression_1$, ..., $expression_n$`);`

La chaîne de contrôle contient le texte à afficher et les spécifications de format correspondant à chaque expression de la liste. Les spécifications de format ont pour but d'annoncer le format des données à visualiser. Elles sont introduites par le caractère %. Le i-ème format de la chaîne de contrôle sera remplacé par la valeur effective de $expression_i$.
Pour être plus précis, une spécification de i-ème format est de la forme :

`%[flag][largeur][.précision][modificateur]type`

Les éléments entre crochet sont facultatifs. Les différents éléments de cette spécification sont les suivants :

1. [**flag**] fournit des options de cadrage (par défaut, $expression_i$ est justifié à droite) et peut prendre les valeurs suivantes :
   - `-` $expression_i$ sera justifié à gauche (ajout de blancs si nécessaire)
   - `+` affichage du signe (+ ou -) avant la valeur numérique
   - *espace* impression d'un espace devant un nombre positif, à la place du signe

   Il existe d'autres valeurs possibles mais qui sont moins utiles.
2. [**largeur**] est le nombre minimal de caractères à écrire (des blancs sont ajoutés si nécessaire). Si le texte à écrire est plus long, il est néanmoins écrit en totalité. En donnant le signe `*` comme largeur, $expression_i$ fournira la largeur (et $expression_{i+1}$ la variable à afficher)
   Exemple : `printf("%*f",14,var)`.
3. [**.précision**] définit, pour les réels, le nombre de chiffres après la virgule (ce nombre doit être inférieur à la largeur). Dans le cas de nombres entiers, ce champ indique le nombre minimal de chiffres désirés (avec l'ajout de 0 sinon), alors que pour une chaîne (%s), elle indique la longueur maximale imprimée (tronquée si trop longue). Comme pour la largeur, on peut écrire `.*` pour que $expression_{i+1}$ désigne la précision à appliquer.
   Valeur par défaut : 6.
4. [**modificateur**] modifie l'interprétation de $expression_i$ et peut valoir `h` (pour `short`), `l` (`long` pour les entiers, `double` pour les réels), ou encore `L` (`long double` pour les réels).
5. **type** précise l'interprétation à donner à $expression_i$. Les valeurs possibles sont détaillées dans la table 2.8

`printf` retourne le nombre de caractères écrits, ou EOF en cas de problème.
Voici plusieurs exemples didactiques :

```
printf("|%d|\n",14);                    |14|
printf("|%d|\n",-14);                   |-14|
printf("|%+d|\n",14);                   |+14|
```

format	conversion en	écriture
%d	`int`	décimale signée
%ld	`long int`	décimale signée
%u	`unsigned int`	décimale non signée
%lu	`unsigned long`	décimale non signée
%o	`unsigned int`	octale non signée
%lo	`unsigned long`	octale non signée
%x	`unsigned int`	hexadécimale non signée
%lx	`unsigned long`	hexadécimale non signée
%f	`double`	décimale virgule fixe
%lf	`long double`	décimale virgule fixe
%e	`double`	décimale notation exponentielle
%le	`long double`	décimale notation exponentielle
%g	`double`	décimale, représentation la plus courte parmi %f et %e
%lg	`long double`	décimale, représentation la plus courte parmi %lf et %le
%c	`unsigned char`	caractère
%s	`char*`	chaîne de caractères

TAB. 2.8 – Les différents formats de la fonction `printf`

```
printf("|%+d|\n",-14);                    |-14|
printf("|% d|\n",14);                     | 14|
printf("|% d|\n",-14);                    |-14|
printf("|%x|\n",0x56ab);                  |56ab|
printf("|%X|\n",0x56ab);                  |56AB|
printf("|%#x|\n",0x56ab);                 |0x56ab|
printf("|%#X|\n",0x56ab);                 |0X56AB|
========================
printf("|%o|\n",14);                      |16|
printf("|%#o|\n",14);                    |016|
========================
printf("|%10d|\n",14);                    |        14|
printf("|%10.6d|\n",14);                 |    000014|
printf("|%.6d|\n",14);                   |000014|
printf("|%*.6d|\n",10,14);               |    000014|
printf("|%*.*d|\n",10,6,14);             |    000014|
========================
printf("|%f|\n",1.234567890123456789e5);     |123456.789012|
printf("|%.4f|\n",1.234567890123456789e5);   |123456.7890|
printf("|%.20f|\n",1.234567890123456789e5);  |123456.78901234567456413060|
printf("|%20.4f|\n",1.234567890123456789e5); |         123456.7890|
========================
printf("|%e|\n",1.234567890123456789e5);     |1.234568e+05|
printf("|%.4e|\n",1.234567890123456789e5);   |1.2346e+05|
printf("|%.20e|\n",1.234567890123456789e5);  |1.23456789012345674564e+05|
printf("|%20.4e|\n",1.234567890123456789e5); |          1.2346e+05|
========================
printf("|%.4g|\n",1.234567890123456789e-5);  |1.235e-05|
printf("|%.4g|\n",1.234567890123456789e5);   |1.235e+05|
```

```
printf("|%.4g|\n",1.23456789012345678e-3);  |0.001235|
printf("|%.8g|\n",1.23456789012345678e5);   |123456.79|
```

2.5.5 La fonction de saisie `scanf`

La fonction `scanf` permet de récupérer les données saisies au clavier, dans le format spécifié. Ces données sont stockées aux adresses spécifiées par les arguments de la fonction `scanf` (on utilise donc l'opérateur d'adressage & pour les variables scalaires). `scanf` retourne le nombre de valeurs effectivement lues et mémorisées (en ommettant les %*). La syntaxe est la suivante :

`scanf("chaîne de contrôle"`, arg_1, ..., arg_n`);`

La chaîne de contrôle indique le format dans lequel les données lues sont converties. Elle ne contient pas d'autres caractères (notamment pas de `\n`). Comme pour `printf`, les conversions de format sont spécifiées par un caractère précédé du signe %. Les formats valides pour la fonction `scanf` diffèrent légèrement de ceux de la fonction `printf`.
Les données à entrer au clavier doivent être séparées par des blancs ou des <RETURN> sauf s'il s'agit de caractères.
Pour être plus précis, les spécifications de format ont la forme suivante :

`%[*][larg][modif]type`

`*`	la valeur sera lue mais ne sera pas stockée
`larg`	précise le nombre maximal de caractères à lire

`modif`	spécifie une taille différente pour les données pointées par l'argument :
	`h` : `short int`
	`l` : `long int` (pour les entier) ou `double` (pour les nombres flottants).
	`L` : `long double`

`type`	spécifie le type de donnée lue et comment il faut le lire.

La plupart des types de `scanf` sont les mêmes que pour `printf` :

Type	**Description**	**Argument requis**
`c`	caractère simple (espace inclu)	`char *`
`d`	entier : nombre optionnellement précédé par un signe	`int *`
`e,E, f,g,G`	Floating point : nombre décimal précédé eventuellement d'un signe et suivi éventuellement du caractère `e` ou `E` et d'un nombre décimal. Exemple :`-732.103` ou `7.12e4`	`float *`
`o`	entier en notation octale.	`int *`
`s`	Chaîne de caractères (jusqu'à la lecture d'un espace)	`char *`
`u`	entier non signé	`unsigned int *`
`x`	entier hexadecimal	`int *`

Exemple :

```
#include <stdio.h>
int main() {
```

```
    int i;
    printf("entrez un entier sous forme hexadecimale i = ");
    scanf("%x",&i);
    printf("i = %d\n",i);
    return 0;
}
```

Les autres fonctions d'entrées/sorties (permettant notamment la manipulation des fichiers) seront abordées dans le chapitre 6.

Chapitre 3

Les pointeurs

Toute variable manipulée dans un programme est stockée quelque part en mémoire centrale. Cette mémoire est constituée d'octets qui sont identifiés de manière univoque par un numéro qu'on appelle *adresse* comme l'illustre la figure 3.1.

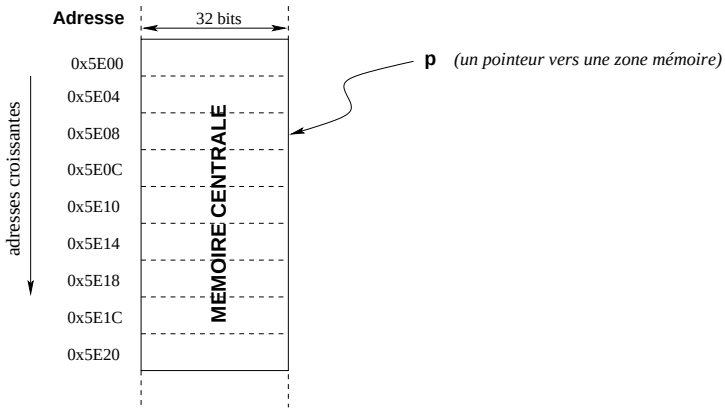


FIG. 3.1 – Illustration de l'adressage de la mémoire centrale

Par analogie, on peut voir la mémoire centrale comme une armoire constituée de tiroirs numérotés. Un numéro de tiroir correspond à une adresse.

Ainsi, lorsqu'on déclare une variable `var` de type `T`, l'ordinateur réserve un espace mémoire (de `sizeof(T)` octets) pour y stocker les valeurs de `var`.
Pour retrouver cette variable, il suffit donc de connaître l'adresse du premier octet où elle est stockée (ou, s'il s'agit d'une variable qui recouvre plusieurs octets contigus, l'adresse du premier de ces octets).

Un pointeur est une variable de type adresse.

Les pointeurs présentent de nombreux avantages :
– Ils permettent de manipuler de façon simple des données de taille importante (comme les tableaux, les structures etc...). Ainsi, au lieu de passer en paramètre à une fonction un élément très grand (en taille), on pourra se contenter

de lui fournir un pointeur vers cet élément... On gagne évidemment alors en efficacité dans l'exécution du programme.
– Comme on le verra au chapitre 4, les tableaux ne permettent de stocker qu'un nombre fixé d'éléments de même type. Si les composantes du tableau sont des pointeurs, il sera possible de stocker des éléments de tailles diverses (comme des chaînes de caractères : voir §4.2.5)
– Il est possible de créer des structures chaînées (on dit aussi structures auto-référées) qui sont utilisées pour définir des listes chainées. De telles listes sont beaucoup utilisées en programmation (le nombre d'éléments de cette liste peut évoluer dynamiquement, ce qui permet une utilisation plus souple que celle des tableaux). Cette notion sera abordée en détail au §4.3.5.

3.1 Déclaration d'un pointeur

En C, chaque pointeur est limité à un type de donnée. En effet, même si la valeur d'un pointeur (une adresse) est toujours un entier (ou éventuellement un entier long), le type d'un pointeur dépend du type de l'objet pointé. Cette distinction est indispensable à l'interprétation (en fait la taille) de la valeur pointée.
On déclare un pointeur par l'instruction :

`type *`*nom-du-pointeur* ;

où `type` est le type de l'objet pointé. Exemple :

```
int *pi;        // pi est un pointeur vers un int
short int *psi; // psi est un pointeur vers un short int
char *pc;       // pc pointeur vers un char
```

A noter aussi l'existence de *pointeurs génériques*, c'est à dire capable de pointer vers n'importe quel type d'objet. On utilise pour cela le type `void *`.
Sans un tel type, il ne serait pas possible par exemple d'indiquer le type d'objet rendu par les fonctions d'allocation de mémoire qui rendent un pointeur vers l'objet alloué, puisque ce type varie d'une invocation à l'autre de la fonction. Par exemple, la fonction `malloc` de la bibliothèque standard est définie de la manière suivante : `void *malloc(size_t size);`

3.2 Opérateurs de manipulation des pointeurs

Lors du travail avec des pointeurs, nous avons besoin :
– d'un opérateur 'adresse de' `&` pour obtenir l'adresse d'une variable.
– d'un opérateur 'contenu de' `*` pour accéder au contenu d'une adresse.

3.2.1 L'opérateur 'adresse de' `&`

L'opérateur `&` permet d'accéder à l'adresse d'une variable. La syntaxe est la suivante :

&nom-variable

Cette adresse peut alors être utilisée pour initialiser la valeur d'un pointeur. Dans l'exemple suivant, on définit un pointeur `p` qui pointe vers un entier `i` :

```
int * p;    //étape (1): pointeur vers un entier non initialisé
int i = 14; //étape (2): variable entière initialisée a 14
p=&i;       //étape (3): p pointe vers i
```

Les différentes étapes de ce scénario sont illustrées dans la figure 3.2.

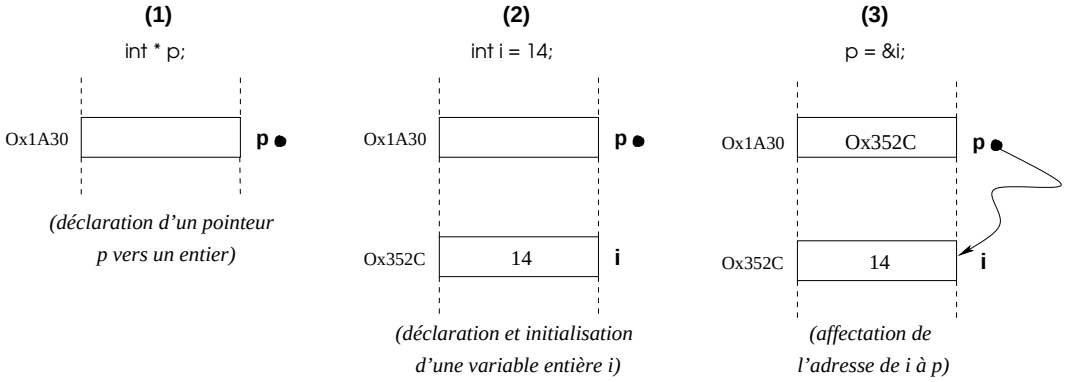


FIG. 3.2 – Illustration mémoire d'une affectation par l'opérateur &

L'opérateur & peut seulement être appliqué à des objets qui se trouvent dans la mémoire interne, c'est à dire à des variables et des tableaux. Il ne peut pas être appliqué à des constantes ou des expressions.

3.2.2 L'opérateur 'contenu de' : *

L'opérateur unaire d'indirection `*` permet d'accéder directement à la valeur de l'objet pointé (on dit qu'on déréférence un pointeur). La syntaxe est la suivante :

**nom-pointeur*

Ainsi, si `p` est un pointeur vers un entier `i`, `*p` désigne la valeur de `i`. Exemple :

```
int main() {
  int i = 14;
  int *p;
  p = &i;                  //p contient l'adresse de i (0x352C)
  printf("*p = %d \n",*p); //affiche "*p = 14"
}
```

Pour résumer, après les instructions précédentes :
– `i` désigne le contenu de `i` (soit `14`)
– `&i` désigne l'adresse de `i` (soit `0x352C`)
– `p` désigne l'adresse de `i` (soit `0x352C`)
– `*p` désigne le contenu de `i` (soit `14`)
En outre :

– `&p` désigne l'adresse de `p` (soit `0x1A30`)
– `*i` est en général illégal (puisque `i` n'est pas un pointeur mais il peut arriver que la valeur `*i` ait un sens)

Petites devinettes (Merci Bernard Cassagne [Cas98]) : Soient `i` et `j` deux pointeurs vers des entiers,

1. A quoi correspond l'expression `*i**j` ?
2. Et `*i/*j` ?

3.3 Initialisation d'un pointeur

Par défaut, lorsque l'on déclare un pointeur `p` sur un objet de type `T`, on ne sait pas sur quoi il pointe. En effet, la case mémoire qu'il occupe contient une certaine valeur qui risque de le faire pointer vers une zone hasardeuse de la mémoire.

Comme toute variable, un pointeur doit être initialisé !

Cette initialisation peut s'effectuer de trois façons :

1. affectation à l'adresse d'une autre variable de `p`. Si la variable est un pointeur, on peut faire l'affectation directement, sinon on doit utiliser l'opérateur `&`. Exemple :

```
int *p1, *p2;//déclaration de 2 pointeurs vers des entiers
int i = 14;  //supposons que i se trouve à l'adresse 0x352C
p1 = &i;     //affectation à l'adresse de i de p1 , soit 0x352C
p2 = p1;     //affectation de p2 à p1:
             //p2 contient aussi l'adresse de i
```

2. affectation de `p` à la valeur `NULL` : on peut dire qu'un pointeur ne pointe sur rien en lui affectant la valeur `NULL` (cette valeur est définie dans le fichier `stddef.h`). Exemple :
   `int * p = NULL;`
3. affectation directe de `*p` (la zone mémoire pointée par `p`). Pour cela, il faut d'abord réserver à `*p` un espace-mémoire de taille adéquate (celui du type pointé par `p`, soit `sizeof(T)` octets). L'adresse de cet espace-mémoire sera la valeur de `p`. Cette opération consistant à réserver un espace-mémoire pour stocker l'objet pointé s'appelle une *allocation dynamique* et sera abordée plus en détail au §3.5.

Pour bien montrer l'intérêt de l'initialisation de tout pointeur, reprenons l'exemple précédent dans lequel on remplace l'affectation `p2 = p1` par `*p2 = *p1` :

```
int * p1, *p2;
int i = 14;
p1 = &i;
*p2 = *p1;
```

Que va t il se passer ?

– `p1` est bien initialisé et pointe sur `i` (`*p1=14`)
– mais `p2` n'a pas été initialisé : `*p2` désigne une adresse mémoire a priori inconnue. L'instruction `*p2 = *p1` force l'écriture de la valeur `*p1=14` dans la case mémoire pointée par `p2` ce qui pourrait avoir des conséquences désastreuses ! Cette ecriture est en général sanctionné par le message d'erreur "`Segmentation fault`" à l'exécution.

Il faut quand même noter que certain compilateurs ont la bonne idée d'initialiser automatiquement à la valeur `NULL` un pointeur non affecté. Mais cela ne doit pas empêcher de prendre la bonne habitude d'initialiser tout pointeur.

3.4 Arithmétique des pointeurs

La valeur d'un pointeur étant un entier, on peut lui appliquer un certain nombre d'opérateurs arithmétiques classiques. Les seules opérations arithmétiques valides sur les pointeurs sont :

– l'addition d'un entier à un pointeur $\longrightarrow$ **p + i**
  Le résultat est un pointeur *de même type* que le pointeur de départ.
– la soustraction d'un entier à un pointeur $\longrightarrow$ **p − i**
  Le résultat est un pointeur *de même type que* le pointeur de départ.
– la différence de deux pointeurs $\longrightarrow$ **p1 − p2**
  Il faut **absolument** que les deux pointeurs pointent vers des objets de même type `T`. Le résultat est un entier dont la valeur est égale à (`p1 - p2`)/`sizeof(T)`.

Attention : la somme de deux pointeurs n'est pas autorisée !

Ainsi, soit `i` un entier et `p` un pointeur sur un objet de type `T` (donc déclaré par l'instruction : `T *p ;`). Alors `p+i` (respectivement `p-i`) désigne un **pointeur sur un objet de type** `T`. Sa valeur est égale à celle de `p` incrémentée (respectivement décrémentée) de `i*sizeof(T)`. Exemple :

```
#include <stdio.h>
int main() {
    int i = 2;
    int *p1, *p2;
    p1 = &i;
    p2 = p1 + 1;
    printf("Adresse p1 = 0x%lx \t Adresse p2 = 0x%lx\n",
            (unsigned long)p1,
            (unsigned long)p2);
    printf("Différence des adresses: %lu \t sizeof(int)=%u\n",
            (unsigned long)p2-(unsigned long)p1,
            sizeof(int));
    printf("MAIS p2-p1 = %i !\n",p2-p1);
    return 0;
}
```

Ce programme renverra :

```
Adresse p1 = 0xbffffb24          Adresse p2 = 0xbffffb28
Différence des adresses: 4       sizeof(int)=4
MAIS p2-p1 = 1 !
```

On peut faire la même expérience avec le type `double` (au lieu du type `int`). On obtiendrait alors :

```
Adresse p1 = 0xbffffb20          Adresse p2 = 0xbffffb28
Différence des adresses: 8       sizeof(double)=8
MAIS p2-p1 = 1 !
```

A noter qu'on peut également utiliser les opérateurs `++` et `--` avec des pointeurs. En général, on les utilise pour réaliser des parcours de tableaux et plus particulièrement dans les chaînes de caractères. Exemple (comme on le verra au §4.2.1, toute chaîne se termine par un caractère *null*, le caractère '\0') :

```
#include <stdio.h>
int main() {
    char * mess = "On est super content!";
    char *p;
    for (p = &mess[0]; *p != '\0'; p++) {
        printf("Adresse: %ld | Contenu: %c\n",(long)p,*p);
    }
    // Autre méthode classique, avec while
    p = mess; // équivalent de p = &mess[0] dans ce cas
    puts("========================================");
    while (*p != '\0') {
        printf("Adresse: %ld | Contenu: %c\n",(long)p,*p);
        p++;
    }
    return 0;
}
```

Les chaînes de caractères seront plus amplement détaillées au §4.2.1 page 43. A noter que les opérateurs de comparaison sont également applicables aux pointeurs[1].

3.5 Allocation dynamique de mémoire

Nous avons vu que l'utilisation de pointeurs permet de mémoriser économiquement des données de différentes grandeurs (puisqu'on se contente de mémoriser l'adresse de ces données). Pour permettre une utilisation efficace de la mémoire, il est primordial de disposer de moyens pour réserver et libérer de la mémoire dynamiquement au fur et à mesure de l'exécution. C'est ce qu'on appelle un *allocation dynamique* de la mémoire. Les fonctions pour la gestion dynamique de la mémoire sont déclarées dans le fichier `stdlib.h` (à inclure).

L'opération consistant à réserver un espace-mémoire est réalisé par la fonction `malloc`. Sa syntaxe est :

`malloc(`*nb_octets*`)`

Cette fonction retourne un pointeur de type `char *` pointant vers une zone mémoire de *nb_octets* octets. Pour initialiser des pointeurs vers des objets qui

[1] à condition bien sûr de comparer des pointeurs qui pointent vers des objets de même type.

ne sont pas de type `char`, il faut convertir le type de la sortie de la fonction `malloc` à l'aide d'un cast (déjà étudié au §2.4).
A noter enfin qu'en pratique, on utilise la fonction `sizeof()` pour déterminer la valeur *nb_octets*. Ainsi, pour initialiser un pointeur vers un entier, on écrit :

```
#include <stdlib.h>
int main() {
    int *p;
    p = (int*)malloc(sizeof(int)); // allocation dynamique
    *p = 14;
}
```

Il est primordial de bien comprendre qu'**avant l'allocation dynamique (et plus généralement avant toute initialisation de `p`), `*p` n'a aucun sens !** En particulier, toute manipulation de la variable `*p` générerait en général une violation mémoire, détectable à l'exécution par le message d'erreur `Segmentation fault`.

La fonction `malloc` permet également d'allouer un espace pour plusieurs objets contigus en mémoire. On peut écrire par exemple

```
#include <stdlib.h>
#include <stdio.h>
int main() {
    int *p;
    p = (int*)malloc(2 * sizeof(int)); //allocation pour 2 int
    *p = 14;
    *(p + 1) = 10;
    printf("p = 0x%lx \t *p = %i \t p+1 = 0x%lx \t *(p+1)=%i\n",
            (unsigned long)p, *p, (unsigned long)(p+1), *(p+1));
    return 0;
}
```

L'appel à la fonction `malloc` a ainsi permis de réserver 8 octets en mémoire (qui permettent de stocker 2 objets de type `int`) et d'affecter à `p` l'adresse de cette zone mémoire. Le programme affiche :
`p = 0x8049718  *p = 14  p+1 = 0x804971c  *(p+1)=10`.

La fonction `calloc` a le même rôle que la fonction `malloc` mais elle permet de réserver *nb-objets* objets de *nb_octets* octets et de les initialiser à zéro. Sa syntaxe est :

`calloc(`*nb-objets*,*nb_octets*`)`

Ainsi, si `p` est de type `int*`, l'instruction
`p = (int*)calloc(N,sizeof(int));`
est sémantiquement équivalente à

```
p = (int*)malloc(N * sizeof(int));
for (i = 0; i < N; i++)
  *(p + i) = 0;
```

L'emploi de `calloc` est simplement plus pratique et plus rapide.

3.6 Libération dynamique avec la fonction `free`

Lorsque l'on n'a plus besoin de l'espace-mémoire alloué dynamiquement par `malloc` ou `calloc` (c'est-à-dire quand on n'utilise plus le pointeur `p` initialisé par ces fontions), il faut libérer ce bloc de mémoire. Ceci se fait à l'aide de l'instruction `free` qui a pour syntaxe :

`free(`*nom-du-pointeur*`);`

Cette instruction libère le bloc de mémoire désigné par *nom-du-pointeur* mais n'a pas d'effet si le pointeur a la valeur `NULL`.
A toute instruction de type `malloc` ou `calloc` doit être associée une instruction de type `free`.
Attention :

- La fonction `free` peut aboutir à un désastre si on essaie de libérer de la mémoire qui n'a pas été allouée par `malloc` ou `calloc`.
- `free` ne change pas le contenu du pointeur.
- Si la mémoire n'est pas libérée à l'aide `free`, alors elle l'est automatiquement à la fin du programme. Cependant, cela ne doit pas dispenser de l'utilisation de cette fonction. L'exemple des listes chaînées (détaillé au §4.3.5) est tout à fait adapté : si la mémoire n'est pas libérée à chaque suppression d'un élément de la liste, on aboutira rapidement à une violation de mémoire et au traditionnel message d'erreur "`Segmentation fault`".

Chapitre 4

Les types dérivés

A partir des types prédéfinis en C (caractères, entiers, flottants), on peut créer de nouveaux types, appelés *types dérivés*, qui permettent de représenter des ensembles de données organisées.

4.1 Les énumérations

Les *énumérations* permettent de définir un type pour des variables qui ne sont affectées qu'a un nombre fini de valeurs.
Un objet de type énumération est défini par le mot-clef `enum` et un identificateur de modèle, suivi de la liste des valeurs que peut prendre cet objet :

`enum modele` $\{constante_1, constante_2, \ldots, constante_n\}$;

En réalité, les objets de type `enum` sont représentés comme des `int`. Les valeurs possibles $constante_1$, $constante_2$,...,$constante_n$ sont codées par des entiers de 0 à n-1.
Dans l'exemple suivant, le type `enum boolean` associe l'entier 0 à la valeur `FALSE` et l'entier 1 à la valeur `TRUE`.

```
#include <stdio.h>
enum boolean {FALSE, TRUE}; //définition de l'enumération boolean
int main () {
    enum boolean b1 = TRUE; //declaration
    printf("b = %d\n",b1);
    return 0;
}
```

On peut modifier le codage par défaut des valeurs de la liste lors de la déclaration du type énuméré, par exemple :
`enum boolean {FALSE = 12, TRUE = 23};`

4.2 Les tableaux

Un tableau est un ensemble fini d'éléments de même type, stockés en mémoire à des adresses contiguës.

La déclaration d'un tableau à une dimension se fait de la façon suivante :

`type nom-du-tableau[nombre-éléments];`

où *nombre-éléments* est une expression constante entière positive correspondant au nombre maximal d'élément dans le tableau. On l'appelle aussi la *dimension* (ou *taille*) du tableau. Chaque élément du tableau est une *composante* de celui-ci.
Par exemple, la déclaration `int tab[10];` indique que `tab` est un tableau de 10 éléments de type `int`. En faisant le rapprochement avec les mathématiques, on dit encore que "t est un vecteur de dimension 10". Cette déclaration alloue donc en mémoire pour l'objet tab un espace de 10*4 octets consécutifs.
Pour plus de clarté, il est recommandé de donner un nom à la constante *nombre-éléments* par une directive au préprocesseur (voir chapitre 7), par exemple :

```
#define TAILLE 20  //noter l'absence de ";"
int t[TAILLE];     //t tableau de TAILLE int
```

Voici les points **importants** à retenir :

1. On accède à un élément du tableau en lui appliquant l'opérateur [].
2. les index des eléments d'un tableau vont de 0 à *nombre-éléments*-1
3. la taille d'un tableau **DOIT** être connue statiquement par le compilateur. Impossible donc d'écrire `int t[n]` ou `n` est une variable.

Ainsi, le programme suivant imprime les éléments du tableau tab :

```
#define N 10
int main() {
   int tab[N];
   int i;
   ...
   for (i = 0; i < N; i++)
      printf("tab[%d] = %d\n",i,tab[i]);
   ...
}
```

Un tableau correspond en fait à un pointeur vers le premier élément du tableau. Ce pointeur est constant, ce qui implique en particulier qu'aucune opération globale n'est autorisée sur un tableau. Notamment, un tableau ne peut pas figurer à gauche d'un opérateur d'affectation : on ne peut pas écrire "`tab1 = tab2;`". Il faut effectuer l'affectation pour chacun des éléments du tableau.

4.2.1 Initialisation d'un tableau

On peut initialiser un tableau lors de sa déclaration par une liste de constantes de la façon suivante :

`type` *nom-du-tableau*[N] = {*constante*$_1$,*constante*$_2$,..., *constante*$_N$};

Par exemple :

```
#define N 5
int t[N] = {14, 27, 3, 18, 81};
int main() {
   int i;
```

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

La figure 4.1 montre ce qu'il se passe en pratique au niveau de la mémoire lors de l'initialisation du tableau `t`. En supposant qu'une variable du type `int` occupe 4 octets (c'est à dire : `sizeof(int)=4`), alors le tableau `t` réservera $N * 4 = 5 * 4 = 20$ octets en mémoire.

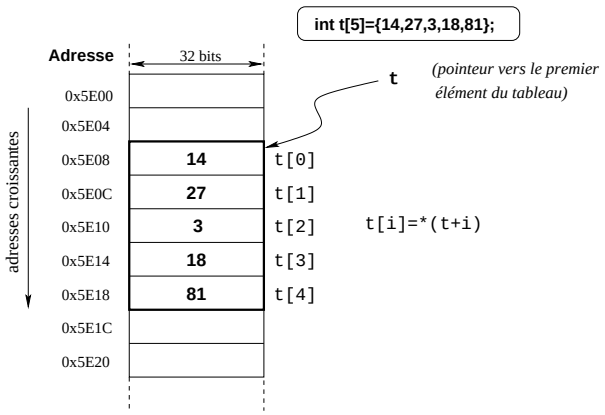


FIG. 4.1 – Expression d'un tableau en mémoire physique

On peut donner moins de valeurs d'initialisations que le tableau ne comporte d'éléments. Dans ce cas, les premiers éléments seront initialisés avec les valeurs indiquées, tandis que les autres seront initalisées à zéro. Exemple :

```
#define N 10
int t[N] = {14,27};
```

Les éléments d'indice 0 et 1 seront initialisés respectivement avec les valeurs 14 et 27, les autres éléments (d'indices 2 à 9) seront initialisés à zéro.

Réservation automatique

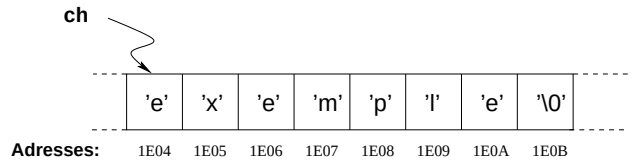
Si la dimension n'est pas indiquée explicitement lors de l'initialisation, alors le compilateur réserve automatiquement le nombre d'octets nécessaires. Exemple :

```
int A[] = {1, 2, 3, 4, 5}; //réservation de 5*sizeof(int) octets
float B[] = {-1.05, 3.3, 87e-5, -1.3E4}; //réservation de
                                         //4*sizeof(float) octets
```

Cas particulier des tableaux de caractères

La représentation interne d'une chaîne de caractères est terminée par le symbole *nul* '\0'.
Ainsi, **pour un texte de n caractères, il faut prévoir $n + 1$ octets**.

– un tableau de caractères peut être initialisé comme une liste de constantes caractères. Exemple : `char ch[3] = {'a', 'b', 'c'};`
Cela devient évidemment très vite lourd.
– On peut aussi initialiser un tableau de caractère par une chaîne littérale : `char ch[8]="exemple";` La figure suivante montre le contenu de la mémoire à la suite de cette initialisation.



– on peut déclarer une taille supérieure à celle de la chaîne littérale :
`char ch[100]="exemple";`
– enfin, on peut ne pas indiquer la taille et le compilateur comptera le nombre de caractères de la chaîne littérale pour dimensionner correctement le tableau (sans oublier le caractère *null* '\0'). Exemple :
`char ch[]="ch aura 22 caractères";`
– Il est également possible de donner une taille égale au nombre de caractères de la chaîne. Dans le cas, le compilateur comprend qu'il ne faut pas rajouter le *null* en fin de chaîne. Exemple : `char ville[10]="luxembourg";`
Mais dans ce cas, attention aux surprises ! Considérer l'exemple suivant :

```
#include <stdio.h>
int main() {
    char t1[10]="luxembourg"; //sans '\0'
    char t2[]="luxembourg";   //avec '\0'
    printf("t1 (de taille %i)=%s\n",sizeof(t1)/sizeof(char),t1);
    printf("t2 (de taille %i)=%s\n",sizeof(t2)/sizeof(char),t2);
    return 0;
}
```

Ce programme renverra :

```
t1 (de taille 10)=luxembourgßüPÅ@ãd@ ûßXAûßÆ-@
t2 (de taille 11)=luxembourg
```

En effet, pour t1, toute la zone mémoire est affichée jusqu'à ce que le caratère nul soit rencontré.

4.2.2 Tableaux multidimensionnels

En C, un tableau multidimensionnel est considéré comme un tableau dont les éléments sont eux-même des tableaux. Un tableau à deux dimensions se déclare donc de la manière suivante :

```
int mat[10][20];
```

En faisant le rapprochement avec les mathématiques, on peut dire que `mat` est une matrice de 10 lignes et de 20 colonnes. Les mêmes considérations que celles que nous avons développé sur les tableaux unidimensionnels s'appliquent :

1. A la déclaration, le compilateur allouera une zone mémoire permettant de stocker de manière contigüe 10 tableaux de 20 entiers, soit 200 entiers ;

2. toute référence ultérieure à `mat` sera convertie en l'adresse de sa première ligne, avec le type pointeur vers un tableau de 20 `int`.

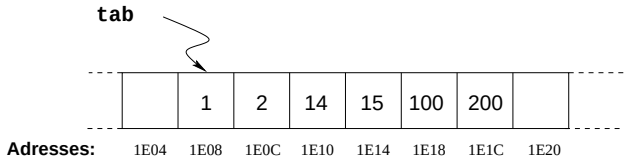
On accède à un élément du tableau par l'expression `mat[i][j]`.
Pour initialiser un tableau à plusieurs dimensions à la compilation, on utilise une liste dont chaque élément est une liste de constantes :

```
#include <stdio.h>
#define L 3   //nombre de lignes
#define C 2   //nombre de colonnes
short tab[L][C] = {{1, 2}, {14, 15}, {100, 200}};
int main() {
   int i, j;
   for (i = 0 ; i < L; i++) {
      for (j = 0; j < C; j++)
        printf("tab[%d][%d]=%d\n",i,j,tab[i][j]);
   }
   return 0;
}
```

On vient de définir la matrice

$$\texttt{tab} = \begin{pmatrix} 1 & 2 \\ 14 & 15 \\ 100 & 200 \end{pmatrix}$$

La figure suivante montre le contenu de la mémoire à la suite de cette initialisation.



On comprend bien avec cette figure que si l'on souhaite utiliser la réservation automatique, il faudra quand même spécifier toutes les dimensions sauf la première (le nombre de lignes dans le cas d'un tableau bi-dimentionnel.

```
short int mat[][3] = {{1, 0, 1},
                      {0, 1, 0}};
```

réservation de 2*3*2 = 12 octets et

$$\texttt{mat} = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

4.2.3 Passage de tableau en paramètre

Puisqu'un identificateur de type tableau correspond à l'adresse du premier élément du tableau (voir figure 4.1), c'est cette adresse qui est passée en paramètre formel. Considérons par exemple une fonction `print_tab` qui affiche le contenu d'un tableau d'entiers :

```
#include <stdio.h>
/* Affiche le contenu du tableau d'entiers tab ayant nb_elem composantes */
void print_tab(int tab[], int nb_elem) {
    int i; //compteur
    for (i=0; i < nb_elem; i++) printf("tab[%i]=%i\n",i,tab[i]);
}
#define TAILLE 4
int main() {
    int t[TAILLE] = {1, 2, 3, 4};
    print_tab(t, TAILLE);
    return 0;
}
```

Dans le prototype d'une fonction, l'écriture `int tab[]` est strictement équivalente à `int * tab` (mais on préferera la première écriture pour des raisons de lisibilité).

Modification des éléments d'un tableau passé en paramètre

Puisque finalement on passe un pointeur en paramètre, on peut modifier au besoin le contenu du tableau. On peut écrire par exemple :

```
#include <stdio.h>
void print_tab(int tab[], int nb_elem) {...}
/* incrémente chaque composantes d tableau */
void incr_tab(int tab[], int nb_elem) {
    int i;
    for (i=0; i < nb_elem; i++) tab[i]++;
}
#define TAILLE 4
int main() {
    int t[TAILLE] = {1, 2, 3, 4};
    incr_tab(t, TAILLE);
    print_tab(t, TAILLE);
    return 0;
}
```

Interdire la modification des éléments d'un tableau passé en paramètre

On utilise pour cela le mot-clé `const` qui spécifie que la variable associée ne peut pas être modifiée[1]. Exemple :

```
const int i = 14; //cette variable ne pourra plus être modifiée
```

On voit tout de suite l'intérêt de `const` pour les paramètres de fonction. Ainsi, dans la procédure `print_tab`, on peut exprimer le fait que cette procédure ne doit pas modifier le contenu du tableau passé en paramètre. Le prototype de la procédure deviendra pour cela :

```
void print_tab(const int tab[], int nb_elem);
```

[1] En pratique, la mémoire n'est pas protégée en écriture et seule l'utilisation directe de la variable constante empêche l'écriture. Il est toujours possible d'écrire à l'adresse de la variable par des moyens détournés.

4.2.4 Relation entre tableaux et pointeurs

Pointeurs et tableaux à une dimension

On rappelle que tout tableau en C est en fait un pointeur constant.
Ainsi, la déclaration `int tab[10];` définit un tableau de 10 valeurs entières dont les indices varient entre les valeurs 0 et 9 et `tab` est un **pointeur constant** (non modifiable) dont la valeur est l'**adresse du premier élément** du tableau. Autrement dit, `tab` a pour valeur `&tab[0]`. On peut donc utiliser un pointeur initialisé à tab pour parcourir les éléments du tableau, comme dans l'exemple suivant :

```
#define N 5
int tab[5] = {1, 2, 6, 0, 7};
int main() {
    int i;
    int *p;
    p = tab;
    for (i = 0; i < N; i++) {
        printf(" %d \n",*p);
        p++;
    }
    return 0;
}
```

Comme on accède à l'élément d'indice `i` du tableau `tab` grâce à l'opérateur d'indexation `[]` par l'expression `tab[i]`, on peut déduire la relation entre cet opérateur d'indexation et l'opérateur `*` :

`tab[i] = *(tab + i)`

Pour résumer, on a les équivalences suivantes :

`tab + 1`	$\Longleftrightarrow$	`&(tab[1])`	*(adresse du 2ième élément)*
`*(tab + 1)`	$\Longleftrightarrow$	`tab[1]`	*(valeur du 2ième élément)*
`*tab`	$\Longleftrightarrow$	`tab[0]`	*(valeur du 1er élément)*
`*(tab + k)`	$\Longleftrightarrow$	`tab[k]`	*(valeur du (k+1)ème élément)*
`tab + k`	$\Longleftrightarrow$	`&(tab[k])`	*(adresse du (k+1)ème élément)*

Pointeurs et tableaux se manipulent donc exactement de même manière. Attention cependant puisqu'aucun contrôle n'est effectué pour s'assurer que l'élément du tableau adressé existe effectivement. En outre, la manipulation de tableaux possède certains inconvénients par rapport à la manipulation des pointeurs dûs au fait qu'un tableau est un pointeur constant :

– On ne peut pas créer de tableaux dont la taille est une variable du programme[2] ;
– on ne peut pas créer de tableaux bidimensionnels dont les lignes n'ont pas toutes le même nombre d'éléments.

Ces opérations deviennent possibles dès que l'on manipule des pointeurs alloués dynamiquement. Ainsi, pour créer un tableau d'entiers à n éléments où n est une variable du programme, on écrit :

```
#include <stdlib.h>
int main() {
```

[2]En fait, on peut le faire depuis la norme C99 mais on ignorera cette possibilité ici.

```
  int n;
  int *tab;
  ...
  tab = (int*)malloc(n * sizeof(int));
  ...
  free(tab);
}
```

Si on veut en plus que tous les éléments du tableau tab soient initialisés à 0, on remplace l'allocation dynamique avec `malloc` par :

```
tab = (int*)calloc(n, sizeof(int));
```

On pourra aussi utiliser la fonction `memset`.
Les éléments de `tab` sont manipulés avec l'opérateur d'indexation `[]`, exactement comme pour les tableaux.
Pour conclure, on retiendra que les deux principales différence entre un tableau et un pointeur sont les suivantes :

- un pointeur doit toujours être initialisé, soit par une allocation dynamique, soit par affectation à une expression de type adresse, par exemple `p = &i ;`
- un tableau est un pointeur constant ne peut donc pas figurer à gauche d'un opérateur d'affectation. En particulier, un tableau ne pas être utilisé directement dans une expression arithmétique (on ne peut pas écrire `tab++ ;` par exemple).

Pointeurs et tableaux à plusieurs dimensions

Un tableau à deux dimensions peut être vu de deux façons :

- soit comme un pointeur sur une zone mémoire de taille le produit des deux dimensions, comme c'est le cas dans l'écriture :

  ```
  int tab[L][C];
  ```

  Avec cette écriture, `tab` a une valeur *constante* égale à l'adresse du premier élément du tableau, c'est à dire `&tab[0][0]`.
- soit comme un tableau de pointeurs. Chaque élément du tableau est alors lui-même un pointeur sur un tableau. On déclare un pointeur qui pointe sur un objet de type `type *` (deux dimensions) de la même manière qu'un pointeur, c'est-à-dire par la déclaration :

  ```
  type **nom-du-pointeur ;
  ```

  De même un pointeur qui pointe sur un objet de type `type **` (équivalent à un tableau à 3 dimensions) se déclare par

  ```
  type ***nom-du-pointeur ;
  ```

L'exemple suivant illustre la déclaration d'un tableau à deux dimensions (une matrice de k lignes et n colonnes à coefficients entiers) sous forme d'un tableau de pointeurs :

```
int main() {
    int k=14, n=5;
    int **tab; // pointeur vers la matrice

    tab = (int**)malloc(k * sizeof(int*));
    for (i = 0; i < k; i++)
        tab[i] = (int*)malloc(n * sizeof(int));
```

```
    ....
    for (i = 0; i < k; i++)
        free(tab[i]);
    free(tab);
    return 0;
}
```

La première allocation dynamique réserve pour l'objet pointé par tab l'espace-mémoire correspondant à `k` pointeurs sur des entiers. Ces `k` pointeurs correspondent aux lignes de la matrice. Les allocations dynamiques suivantes réservent pour chaque pointeur `tab[i]` l'espace-mémoire nécessaire pour stocker `n` entiers. Si on désire en plus que tous les éléments du tableau soient initialisés à 0, il suffit d'utiliser la fonction `calloc` au lieu de `malloc` (voir §3.5 page 38)
L'un des avantages des pointeurs de pointeurs sur les tableaux multi-dimensionnés est que cela permet de choisir par exemple des tailles différentes pour chacune des lignes `tab[i]`.

4.2.5 Cas des tableaux de chaînes de caractères

On peut initialiser un tableau de ce type par :

```
char * jour[] = {"lundi", "mardi", "mercredi", "jeudi",
                 "vendredi", "samedi", "dimanche"};
```

L'allure du tableau `jour` est illustré dans la figure 4.2.

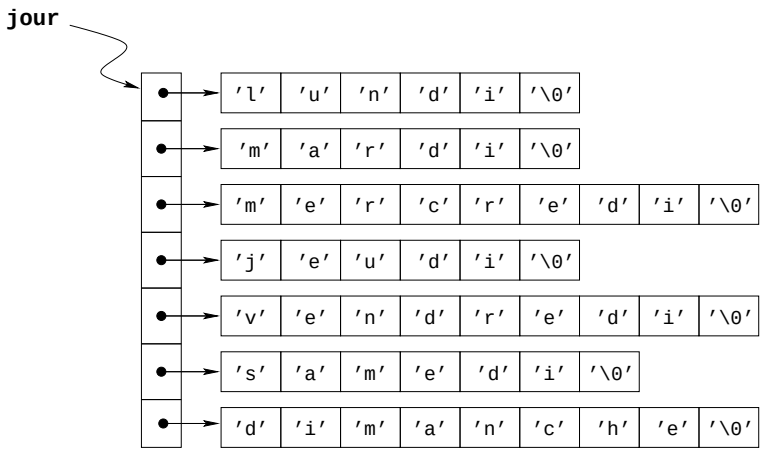


FIG. 4.2 – Illustration de la déclaration d'un tableau de chaînes de caractères

A noter que cette écriture directe n'est possible qu'avec les `char`. Il est impossible d'écrire par exemple :

```
int * tab[] = {{1}, {1,2}, {1,2,3}};
```

(on a ici un tableau de pointeurs vers des tableaux d'entiers de taille différentes).
Une boucle d'impression des valeurs du tableau jour pourra s'écrire :

```
#define NB_JOUR 7
int i;
for (i=0 ; i < NB_JOUR; i++) printf("%s\n",jour[i]);
```

4.2.6 Gestion des arguments de la ligne de commande

Les tableaux de pointeurs vers des chaînes de caractères sont une structure de donnée très importante puisqu'elle est utilisée dans la transmission de paramètres lors de l'exécutution d'un programme.
Lorsqu'un utilisateur lance l'exécution du programme `prog` avec les paramètres `param_1`, `param_2`, ..., `param_n`, l'interpréteur collecte tous ces mots sous forme de chaîne de caractères, crée un tableau de pointeurs vers ces chaînes et lance la procédure `main` en lui passant deux paramètres :
– un entier contenant la taille du tableau (appelé classiquement `argc`) ;
– le tableau des pointeurs vers les chaînes (traditionnellement `argv`).
Pour que le programmeur puisse exploiter ces éléments, la fonction `main` doit être définie de la façon suivante :
`int main(int argc, char * argv[]) {...}`
Voici un exemple d'utilisation de ces paramètres :

```
#include <stdio.h>
int main(int argc, char * argv[]) {
    int i;
    printf("Nom du programme: %s\n", argv[0]);
    for (i=1 ; i < argc ; i++)
        printf("Paramètre %i: %s\n",i,argv[i]);
    return 0;
}
```

En compilant ce programme par la commande `gcc -Wall -g3 toto.c`, l'appel `./a.out nif naf nouf`
renvoit :

```
Nom du programme: ./a.out
Paramètre 1: nif
Paramètre 2: naf
Paramètre 3: nouf
```

La librairie `getopt.h` permet également de gérer plus simplement les arguments de la ligne de commande. Supposons par exemple vouloir compiler un programme `prog` au format d'appel suivant (les éléments entre crochets sont optionnels) : `prog [-i val_i] [-f val_f] [-s string] [-k key] [-h] file`
sachant que :

1. l'option `-i` permet de spécifier la valeur de la variable `val_i`, de type `int` (14 par défaut) ;
2. l'option `-f` permet de spécifier la valeur de la variable `val_f`, de type `float` (1.5 par défaut).
3. l'option `-s` permet de spécifier la valeur de la variable `string`, une chaîne de caractères possédant au plus 256 caractères ("On est super content !" par défaut).
4. l'option `-k` permet de spécifier au format hexadécimal la valeur de la variable `key`, de type `unsigned long` (`0x90ABCDEF` par défaut).

5. l'option `-h` affiche l'aide (auteur, but du programme et usage) et sort du programme.
6. `file` est un paramètre obligatoire du programme qui spécifie la valeur de la chaîne de caractères `input_file`.

Pour gérer ces options, on peut soit adapter l'exemple de code précédent et traiter individuellement les éléments du tableau `argv` ainsi que les cas d'erreurs, soit utiliser la librairie `<getopt.h>`[3] qui permet de gérer facilement les options de la ligne de commande.
Cet exemple est traité avec `<getopt.h>` en annexe A.1 page 106.

4.3 Les structures

Une *structure* est une suite finie d'objets de types différents. Ce mécanisme permet de grouper un certain nombre de variables de types différents au sein d'une même entité. Contrairement aux tableaux, les différents éléments d'une structure n'occupent pas nécessairement des zones contiguës en mémoire. Chaque élément de la structure, appelé *membre* ou *champ*, est désigné par un identificateur. L'utilisation pratique d'une structure se déroule de la façon suivante :

1. On commence par déclarer la structure elle-même. Le modèle général de cette déclaration est le suivant :
   `struct` *nom_structure* {
   `type_1` $membre_1$;
   `type_2` $membre_2$;
   ...
   `type_n` $membre_n$;
   } ;
2. Pour déclarer un objet de type structure correspondant au modèle précédent, on utilise la syntaxe :
   `struct` *nom_structure* identificateur-objet ;
   Ou bien, si le modèle n'a pas encore été déclaré au préalable :
   `struct` *nom_structure* {
   `type_1` $membre_1$;
   `type_2` $membre_2$;
   ...
   `type_n` $membre_n$;
   } identificateur-objet ;
3. On accède aux différents membres d'une structure grâce à l'opérateur membre de structure, noté ".". Le i-ème membre de objet est désigné par l'expression :
   identificateur-objet.$membre_i$

On peut effectuer sur le i-ème membre de la structure toutes les opérations valides sur des données de type `type_i`.

[3] informations : `man 3 getopt` ou plus simplement le tutorial disponible à l'adresse : `http://www.stillhq.com/extracted/howto-getopt/output.ps`

Ainsi, le programme suivant définit la structure `complexe`, composée de deux champs de type double et calcule la norme d'un nombre complexe.

```
#include <math.h>
struct complexe {
  double reelle;     //partie réelle
  double imaginaire; //partie imaginaire
};  // ne pas oublier le ";"

int main() {
  struct complexe z; //déclaration d'un objet de type struct complexe
  double norme;
   ...
  norme = sqrt(z.reelle * z.reelle + z.imaginaire * z.imaginaire);
  printf("norme de (%f + i %f) = %f \n",z.reelle,z.imaginaire,norme);
}
```

4.3.1 Initialisation et affectation d'une structure

Les règles d'initialisation d'une structure lors de sa déclaration sont les mêmes que pour les tableaux. On écrit par exemple :

```
struct complexe i = {0. , 1.};
```

A noter qu'a la différence des tableaux, on peut appliquer l'opérateur d'affectation aux structures. Dans le contexte précédent, on peut écrire :

```
int main() {
  struct complexe z1, z2;
   ...
  z2 = z1;
}
```

4.3.2 Comparaison de structures

Aucune comparaison n'est possible sur les structures (en particulier, on ne peut appliquer les opérateurs `==` ou `!=`).

4.3.3 Tableau de structures

On déclare un tableau dont les composantes sont des structures de la même façon qu'on déclarerait un tableau d'éléments de type simple. Exemple :

```
struct personne {
   char nom[20];
   char prenom[20];
};
...
struct personne t[100]; //tableau de 100 personnes
```

Pour référencer le nom de la personne qui a l'index i, on écrira :

```
t[i].nom
```

4.3.4 Pointeur vers une structure

En reprenant la structure `personne` précédentes, on déclarera une variable de type pointeur vers cette structure de la manière suivante :

```
struct personne *p;
```

On peut affecter à ce pointeur des adresses sur des `struct personne`. Exemple :

```
struct personne {...};
int main() {
   struct personne pers; // pers = variable de type struct personne
   struct personne * p;  // p = pointeur vers une struct personne
   p = &pers;  // p pointe maintenant vers pers
}
```

Pour accèder aux membres de la structure pointée :
– il faudrait écrire normalement `(*p).nom`[4] ;
– mais on utilise en pratique l'écriture simplifiée `p->nom`.

4.3.5 Structures auto-référées

Il s'agit de cas particulier de structures dont un des membres pointe vers une structure du même type. Cette représentation permet en particulier de construire des *listes chaînées*.
En effet, il est possible de représenter une liste d'éléments de même type par un tableau (ou un pointeur). Toutefois, cette représentation, dite contiguë, impose que la taille maximale de la liste soit connue a priori (on a besoin du nombre d'éléments du tableau lors de l'allocation dynamique). Pour résoudre ce problème, on utilise une représentation chaînée : l'élément de base de la chaîne est une structure appelée cellule qui contient la valeur d'un élément de la liste et un pointeur sur l'élément suivant. Le dernier élément pointe sur le pointeur `NULL` (défini dans `stddef.h`). La liste est alors définie comme un pointeur sur le premier élément de la chaîne.
Considérons par exemple la structure `toto` possédant 2 champs :

1. un champ `data` de type `int` ;
2. un champ `next` de type pointeur vers une `struct toto`.

La liste est alors un objet de type pointeur sur une `struct toto`. Grâce au mot-clef `typedef`, on peut définir le type `liste`, synonyme du type pointeur sur une `struct toto` (voir §4.6). On peut alors définir un objet `liste l` qu'il convient d'initialiser à `NULL`. Cette représentation est illustrée dans la figure 4.3.
Un des avantages de la représentation chaînée est qu'il est très facile d'insérer un élément à un endroit quelconque de la liste. Ainsi, pour insérer un élément en tête de liste, on utilise la fonction suivante :

[4] et non `*p.nom` qui sera interprété (compte tenu des priorités entre opérateurs) par `*(p.nom)` ce qui n'a de sens que si le champ `nom` est un pointeur !

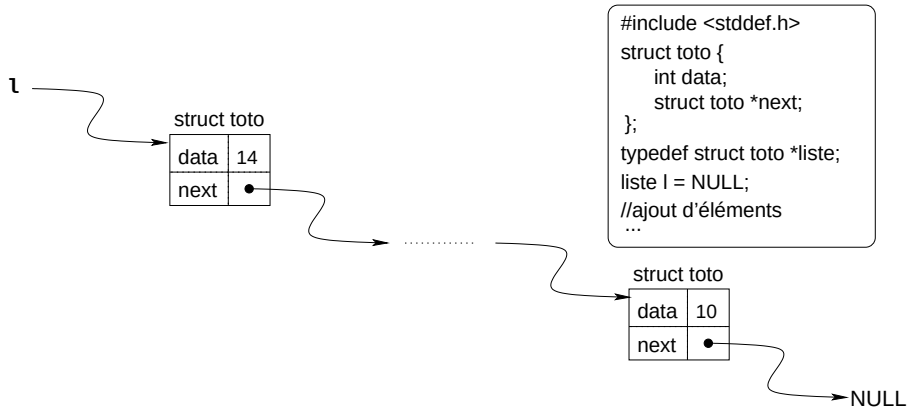


FIG. 4.3 – Representation d'une liste chaînée

```
liste insere(int element, liste Q) {
    liste L;
    L = (liste)malloc(sizeof(struct toto)); // allocation de la zone
                                            // mémoire necéssaire
    L->data = element;
    L->next = Q;
    return L;
}
```

De façon général, l'ajout d'un nouvel élément dans une liste chaînée s'effectue en trois étapes :

1. recherche de l'endroit où la nouvelle cellule devra être insérée ;
2. création de la nouvelle cellule (par `malloc`) et mise à jour de ses champs. Au niveau de l'allocation en mémoire, on prendra garde :
   (a) de réserver le bon nombre d'octets (en reprenant l'exemple précédent, `sizeof(struct toto)` et non `sizeof(struct toto *)`)
   (b) de convertir le résultat du `malloc` vers le bon type (pointeur vers la structure).
3. mise à jour des champs des cellules voisines.

Supposons par exemple que l'on souhaite insérer un élément en queue de liste :

```
liste insereInTail(int element, liste Q) {
    liste L, tmp=Q;
    // creation de la nouvelle cellule
    L = (liste)malloc(sizeof(struct toto)); // allocation de la zone
                                            // mémoire necéssaire
    L->data = element;
    L->next = NULL;
    if (Q == NULL)
        return L;
    // maintenant Q est forcement non vide ==> champ tmp->next existe
    // recherche de l'endroit ou insérer
    while (tmp->next != NULL) tmp = tmp->next; // déplacement jusqu'au
                                        // dernier élément de la liste
```

```
    // mise à jour des cellules voisines (ici, une seule: tmp)
    tmp->next = L;
    return Q;
}
```

Pour supprimer le premier élément de la liste par exemple, il est important de libérer l'espace mémoire alloué :

```
liste supprime_tete(liste L) {
    liste suivant = L;
    if (L != NULL) { // pour etre sûr que L->next existe
       suivant= L->next;
       free(L); //libération de l'espace alloué pour une cellule
    }
    return suivant;
}
```

Il est **primordial** d'utiliser `free` dans le cas des listes chaînées. Sinon, la mémoire risque d'être rapidement saturée (par exemple lors d'ajouts et de suppressions successives qui ne libèrent pas l'espace alloué). Ainsi, de façon similaire à l'ajout, la suppression d'une cellule dans une liste chaînée s'effectue en trois étapes :

1. Recherche de la cellule à supprimer ;
2. mise à jour des champs des cellules voisines ;
3. libération de la cellule avec `free`.

Un exemple plus complet de l'utilisation des listes chaînées est founi en annexe A.2 page 110.

4.4 Les unions

Il est parfois nécessaire de manipuler des variables auxquelles on désire affecter des valeurs de types différents. Une *union* désigne un ensemble de variables de types différents susceptibles d'occuper alternativement une même zone mémoire. Une union permet donc de définir un objet comme pouvant être d'un type au choix parmi un ensemble fini de types. Si les membres d'une union sont de longueurs différentes, la place réservée en mémoire pour la représenter correspond à la taille du membre le plus grand.

4.4.1 Déclaration d'une union

La déclaration et la définition d'une union ne diffèrent de celles d'une structure que par l'utilisation du mot-clé `union` (qui remplace le mot-clé `struct`).
Dans l'exemple suivant, la variable `hier` de type `union jour` peut être soit un caractère, soit un entier (mais pas les deux à la fois) :

```
#include <stdio.h>
union jour {
    char lettre;
    int numero;
```

```
};
int main() {
    union jour hier, demain;
    hier.lettre = 'J'; //jeudi
    printf("hier = %c\n",hier.lettre);
    hier.numero = 4;
    demain.numero = (hier.numero + 2) % 7;
    printf("demain = %d\n",demain.numero);
    return 0;
}
```

Si on se réfère au tableau 1.4 page 12, la zone mémoire allouée pour une variable de type `union jour` sera de `sizeof(int)` (2 ou 4 octets).
On aura compris qu'on accède aux éléments d'une union avec le même opérateur de sélection (`.` ou `->`) que celui utilisé dans les structures.

4.4.2 Utilisation pratique des unions

Lorsqu'il manipule des unions, le programmeur n'a malheureusement aucun moyen de savoir à un instant donné quel est le membre de l'union qui possède une valeur.
Pour être utilisable, une union doit donc toujours être associée à une variable dont le but sera d'indiquer le membre de l'union qui est valide. En pratique, une union et son indicateur d'état sont généralement englobés à l'intérieur d'une structure. Exemple :

```
#include <stdio.h>
enum etat {CARAC, ENTIER};
/* struct jour a deux membres 'indicateur' de type int,
   et 'day' de type union de char et de int */
struct jour {
    enum etat indicateur; //indique ce qui est dans j
    union {
        char lettre;
        int numero;
    } day;
};
/* Affiche le contenu de la variable d, nommée name */
void print_jour(struct jour d, char * name){
    if (d.indicateur == CARAC) printf("%s = %c\n",name,d.day.lettre);
    else if (d.indicateur == ENTIER)
          printf("%s = %i\n",name,d.day.numero);
    else printf("Erreur!\n");
}

int main() {
    struct jour ex1, ex2; //déclaration
    //utilisation
    ex1.indicateur = CARAC;
    ex1.day.lettre = 'j'; //jeudi
    print_jour(ex1, "ex1");
    ex2.indicateur = ENTIER;
    ex2.day.numero = 4;
```

```
    print_jour(ex2, "ex2");
    return 0;
}
```

L'exécution de ce programme renverra :
`ex1 = j`
`ex2 = 4`

4.4.3 Une méthode pour alléger l'accès aux membres

Quand une union est dans une structure (comme c'est la cas dans l'exemple précédent), on aura noter que l'accès aux membres de l'union est relativement lourd (ex : `d.day.numero` pour accéder au membre `numero`). On peut alléger cette écriture en utilisant les facilités du préprocesseur (voir le chapitre 7) :

```
#define L day.lettre
#define N day.numero
...
ex1.indicateur = CARAC;
ex1.L = 'j'; //initialisation de ex1 à jeudi
```

4.5 Les champs de bits

Il est possible en C de spécifier la longueur des champs d'une structure au bit près si ce champ est de type entier (`int` ou `unsigned int`). Cela se fait en précisant le nombre de bits du champ avant le ";" qui suit sa déclaration :

`type` [*membre*] : `nb_bits` ;

On utilise typiquement les champs de bits en programmation système, pour manipuler des registres particuliers de la machine etc... Par exemple, imaginons le registre suivant :

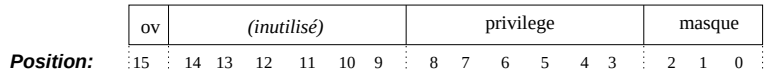


FIG. 4.4 – Exemple d'affectation de bits dans un registre

Ce registre peut se décrire de la manière suivante :

```
struct  registre {
      unsigned int masque : 3;
      signed   int privilege : 6;
      unsigned int : 6;           /* inutilisé */
      unsigned int ov : 1;
};
```

Le champ `masque` sera codé sur 3 bits, `privilege` sur 6 bits etc... A noter que l'ordre dans lequel les champs sont placés à l'intérieur de ce mot dépend de l'implémentation. On voit que le C accepte que l'on ne donne pas de nom

aux champs de bits qui ne sont pas utilisés. Le champ `ov` de la structure ne peut prendre que les valeurs 0 ou 1. Aussi, si r est un objet de type `struct registre`, l'opération `r.ov += 2 ;` ne modifie pas la valeur du champ.
Voici les quelques contraintes à respecter :
– La taille d'un champ de bits doit être inférieure au nombre de bits d'un entier (long).
– un champ de bits n'a pas d'adresse ; on ne peut donc pas lui appliquer l'opérateur `&`.

4.6 Définition de synonymes de types avec `typedef`

Pour alléger l'écriture des programmes, on peut affecter un nouvel identificateur à un type à l'aide du mot-clé `typedef` :

`typedef type` *synonyme* ;

Exemple :

```
typedef unsigned char UCHAR;
typedef struct { double x, y } POINT;
typedef POINT *P_POINT; //pointeur vers un POINT
```

A partir de ce moment, l'identificateur `UCHAR` peut être utilisé comme une abbréviation pour le type `unsigned char`, l'identificateur `POINT` peut être utilisé pour spécifier le type de structure associé et `P_POINT` permet de caractériser un pointeur vers un `POINT` :

```
UCHAR  c1, c2, tab[100];
POINT point;
P_POINT pPoint;
```

Chapitre 5

Retour sur les fonctions

La structuration de programmes en sous-programmes se fait en C à l'aide de *fonctions*. L'utilisation de fonctions a déjà été vu, par exemple avec celle prédéfinies dans des bibliothèques standards (comme `printf` de `<stdio.h>`, `malloc` de `<stdlib.h>` etc...) ou encore la fonction d'entrée `main`. Lorsque les listes chaînées ont été abordées au §4.3.5, on a également défini des fonctions d'insertion et de suppression d'éléments.

On l'aura compris : comme dans la plupart des langages, on peut (doit ?) en C découper un programme en plusieurs fonctions. Il existe une unique fonction obligatoire : la fonction principale appelée `main`.

5.1 Déclaration et définition d'une fonction

Comme on l'a vu au §1.4, le format général d'une définition de fonction est de la forme :

`type_resultat` *nom_fonction* `(`$type_1\ arg_1, \ldots, type_n\ arg_n$`) {`
<déclaration de variables locales >
<liste_d'instructions >
`}`

Dans cette définition de fonction, on rappelle que :
- `type_resultat` correspond au type du résultat de la fonction.
- *nom_fonction* est le nom qui identifie la fonction.
- $type_1\ arg_1\ \ldots type_n\ arg_n$ définit les types et les noms des paramètres de la fonction. Ces arguments sont appelés *paramètres formels*, par opposition aux *paramètres effectifs* qui sont les paramètres avec lesquels la fonction est effectivement appelée (voir §5.2).

On peut se contenter de déclarer le **prototype** d'une fonction (sans le corps) :
`type_resultat` *nom_fonction* `(`$type_1\ arg_1, \ldots, type_n\ arg_n$`) ;`

Dans ce cas, la définition de la fonction (avec le corps des instructions qui la

compose) peut être déclarée plus loin dans le programme[1].
Contrairement à la définition de la fonction, le prototype n'est donc pas suivi du corps de la fonction (contenant les instructions à exécuter).
ATTENTION ! Le prototype est une instruction, il est donc suivi d'un point-virgule !
Quelques remarques complémentaires :

1. Une fonction peut fournir comme résultat :
   - un type arithmétique,
   - une structure (voir §4.3),
   - une union (voir §4.4),
   - un pointeur (voir chapitre 3),
   - `void` (voir ci-après).

   Une fonction ne peut pas fournir comme résultat des tableaux, des chaînes de caractères ou des fonctions, *mais* il est cependant possible de renvoyer un pointeur sur le premier élément d'un tableau ou d'une chaîne de caractères.
2. Si une fonction ne fournit pas de résultat, il faut indiquer `void` comme type du résultat.
   Ex : `void  print_liste(liste L){...}`
3. Si une fonction n'a pas de paramètres, on peut déclarer la liste des paramètres comme `(void)` ou simplement comme `()`.
   Ex : `int main(){...} // equivalent a 'int main(void){...}'`
4. Il est interdit de définir des fonctions à l'intérieur d'une autre fonction (comme en Pascal).
5. En principe, l'ordre des définitions de fonctions dans le texte du programme ne joue pas de rôle, mais chaque fonction doit être déclarée ou définie avant d'être appelée.

Dans le cas où `type_resultat` est différent du type `void`, le corps de la fonction **doit** obligatoirement comporter une instruction `return`, dite *instruction de retour à la fonction appelante*. La syntaxe de cette instruction est :

`return` *expression* ;

La valeur de *expression* correspond à la valeur que retourne la fonction et doit donc être de type `type_resultat`.
Plusieurs instructions `return` peuvent apparaître dans une fonction. Le retour au programme appelant sera alors provoqué par le premier `return` rencontré lors de l'exécution (voir la fonction `puissance` de l'exemple qui suit). Ex :

```
/***Renvoit le produit de 2 entiers ***/
int produit (int a, int b) {
  return(a*b);
}
/*** Renvoit la valeur de a^n (version naïve) ***/
int puissance (int a, int n) {
```

[1] mais cette définition doit respecter la déclaration du prototype : n'hésiter pas à utiliser le copier/coller du prototype !

```
  if (n == 0) return 1;
  return a*puissance(a, n-1); //noter l'absence du 'else'
}
```

Enfin, il faut noter que les variables éventuellement déclarées au sein d'une fonction seront locales à celle-ci et n'auront de sens que dans le corps de la fonction. De même, les identificateurs des arguments de la fonction (arg_1,...,arg_n) n'ont d'importance qu'à l'intérieur de celle-ci.

5.2 Appel d'une fonction

L'appel d'une fonction se fait par l'expression :

$$nom\_fonction \texttt{ (} arg_1, \ldots, arg_n \texttt{)}$$

L'ordre et le type des paramètres effectifs de la fonction doivent concorder avec ceux donnés dans la définition de l'en-tête de la fonction. Les paramètres effectifs peuvent être des expressions[2].
De plus, l'ordre d'évaluation des paramètres effectifs n'est pas assuré et dépend du compilateur. Il est donc déconseillé, pour une fonction à plusieurs paramètres, de faire figurer des opérateurs d'incrémentation ou de décrémentation (`++` ou `--`) dans les expressions définissant les paramètres effectifs.

5.3 Durée de vie des identificateurs

Les variables manipulées dans un programme C ne sont pas toutes traitées de la même manière. En particulier, elles n'ont pas toutes la même durée de vie. On distingue deux catégories de variables.

1. Les *variables permanentes* (ou statiques) Une variable permanente occupe un emplacement en mémoire qui reste le même durant toute l'exécution du programme. Cet emplacement est alloué une fois pour toutes lors de la compilation. La partie de la mémoire contenant les variables permanentes est appelée segment de données. Par défaut, les variables permanentes sont initialisées à zéro par le compilateur. Elles sont caractérisées par le mot-clef `static`.
2. Les *variables temporaires* se voient allouer un emplacement en mémoire de façon dynamique lors de l'exécution du programme. Elles ne sont pas initialisées par défaut. Leur emplacement en mémoire est libéré par exemple à la fin de l'exécution d'une fonction secondaire.

Par défaut, les variables temporaires sont situées dans la partie de la mémoire appelée segment de pile. Dans ce cas, la variable est dite automatique. Le spécificateur de type correspondant, `auto`, est rarement utilisé puisqu'il ne s'applique qu'aux variables temporaires qui sont automatiques par défaut.

[2]La virgule qui sépare deux paramètres effectifs est un simple signe de ponctuation ; il ne s'agit pas de l'opérateur virgule.

Une variable temporaire peut également être placée dans un registre de la machine. Un registre est une zone mémoire sur laquelle sont effectuées les opérations machine. Il est donc beaucoup plus rapide d'accéder à un registre qu'à toute autre partie de la mémoire. On peut demander au compilateur de ranger une variable très utilisée dans un registre, à l'aide de l'attribut de type `register`.
Le nombre de registres étant limité, cette requête ne sera satisfaite que s'il reste des registres disponibles. Cette technique permettant d'accélérer les programmes a aujourd'hui perdu tout son intérêt. Grâce aux performances des optimiseurs de code intégrés au compilateur (cf. options `-O` de `gcc` : voir §1.1.4), il est maintenant plus efficace de compiler un programme avec une option d'optimisation que de placer certaines variables dans des registres.
La durée de vie des variables est liée à leur portée, c'est-à-dire à la portion du programme dans laquelle elles sont définies.

5.4 Portée des variables

Selon l'endroit où on déclare une variable, celle-ci pourra être accessible (visible) partout dans le code ou seulement dans une partie de celui-ci (à l'intérieur d'une fonction typiquement), on parle de la *portée* (ou visibilité) de la variable.
Lorsqu'une variable est déclarée dans le code même, c'est-à-dire à l'extérieur de toute fonction ou de tout bloc d'instruction, elle est accessible de partout dans le code (n'importe quelle fonction du programme peut faire appel à cette variable). On parle alors de variable *globale*.
Lorsque l'on déclare une variable à l'intérieur d'un bloc d'instructions (entre des accolades), sa portée se limite à l'intérieur du bloc dans lequel elle est déclarée. On parle de variable *locale*.

5.4.1 Variables globales

Un variable globale est donc une variable déclarée en dehors de toute fonction, au début du fichier, à l'extérieur de toutes les fonctions et sont disponibles à toutes les fonctions du programme. En général, les variables globales sont déclarées immédiatement derrière les instructions `#include` au début du programme. Elles sont systématiquement permanentes.
Remarque : Les variables déclarées au début de la fonction principale `main` ne sont pas des variables globales, mais locales à `main` !
Exemple :
La variable `STATUS` est déclarée globalement pour pouvoir être utilisée dans les procédures A et B.

```
#include <stdio.h>
int STATUS;

void A(...){
    ...
    if (STATUS>0)
        STATUS--;
```

```
    else
        ...
    ...
}

void B(...) {
    ...
    STATUS++;
    ...
}
```

Conseils :
- Les variables globales sont à utiliser avec précaution, puisqu'elles créent des liens invisibles entre les fonctions. La modularité d'un programme peut en souffrir et le programmeur risque de perdre la vue d'ensemble.
- Il faut faire attention à ne pas cacher involontairement des variables globales par des variables locales du même nom. (voir section suivante)
- **Je vous conseille donc d'écrire nos programmes aussi 'localement' que possible.**

5.4.2 Variables locales

Les variables déclarées dans un bloc d'instructions sont uniquement visibles à l'intérieur de ce bloc. On dit que ce sont des variables locales à ce bloc.
Exemple : la variable `nom` est définie localement dans le bloc extérieur de la fonction `hello`. Ainsi, aucune autre fonction n'a accès à la variable nom :

```
void hello() {
    char nom[20];
    printf("Introduisez votre nom : ");
    scanf("%s",&nom);
    printf("Bonjour %s !\n", nom);
}
```

Attention ! *Une variable déclarée à l'intérieur d'un bloc cache toutes les variables du même nom des blocs qui l'entourent.* Exemple :

```
int X; //déclaration d'une variable globale
int fonction(int A) {
    double X; //variable locale qui cache la variable X globale
    ...
}
```

Remarque : bien qu'il soit possible en C[3] de déclarer des variables à l'intérieur d'une boucle ou d'un bloc conditionnel, il est déconseillé de le faire et toutes les déclarations locales devront se faire au début des fonctions.

[3] depuis la norme C99 en fait

5.5 Récursivité

Les définitions par récurrence sont assez courantes en mathématiques. Par exemple, on peut considérer la fameuse suite de Fibonacci :

$$\begin{cases} u_0 = 0 \\ u_1 = 1 \\ u_n = u_{n-1} + u_{n-2} \quad \forall\ n \geq 2 \end{cases}$$

Le fonctions définies en C ont la propriété de récursivité, c'est à dire la capacité de s'appeler elle-même.
Tous les détails de la récursivité ont été fournis au §2.3 page 24.

5.6 Passage de paramètres à une fonction

Les paramètres ou arguments sont les 'boîtes aux lettres' d'une fonction. Elles acceptent les données de l'extérieur et déterminent les actions et le résultat de la fonction.

5.6.1 Généralités

En ce qui concerne le passage de paramètres à une procédure, le programmeur a deux besoins fondamentaux :
– soit il désire passer une valeur qui sera exploitée par l'algorithme de la procédure (c'est ce dont on a besoin quand on écrit par exemple `sin(x)`). Une telle façon de passer un paramètre s'appelle du *passage par valeur* ;
– soit il désire passer une référence à une variable, de manière à permettre à la procédure de modifier la valeur de cette variable. C'est ce dont on a besoin quand on écrit une procédure réalisant le produit de deux matrices `prodmat(a,b,c)` où l'on veut qu'en fin d'exécution, la matrice `c` soit égale au produit matriciel des matrices `a` et `b`. `prodmat` a besoin des valeurs des matrices `a` et `b`, et d'une référence vers la matrice `c`. Une telle façon de passer un paramètre s'appelle du *passage par adresse*.

Conversion automatique

Lors d'un appel, le nombre et l'ordre des paramètres doivent nécessairement correspondre aux indications de la déclaration de la fonction. Les paramètres sont automatiquement convertis dans les types de la déclaration avant d'être passés à la fonction.
Exemple : Le prototype de la fonction `pow` (bibliothèque `<math.h>`) est déclaré comme suit :

```
double pow(double x, double y);
```

Au cours des instructions,

```
int A, B;
...
A = pow (B, 2);
```

On assiste à trois conversions automatiques : avant d'être transmis à la fonction, la valeur de B est convertie en `double` ; la valeur 2 est convertie en 2.0 . Comme `pow` est du type `double`, le résultat de la fonction doit être converti en `int` avant d'être affectée à A.

5.6.2 Passage des paramètres par valeur

En C, le passage des paramètres se fait **toujours** par valeur, autrement dit les fonctions n'obtiennent que les valeurs de leurs paramètres et n'ont pas d'accès aux variables elles-mêmes.
Les paramètres d'une fonction sont à considérer comme des variables locales qui sont initialisées automatiquement par les valeurs indiquées lors d'un appel. A l'intérieur de la fonction, On peut donc changer les valeurs des paramètres sans influencer les valeurs originales dans les fonctions appelantes.
Exemple :
La fonction `ETOILES` dessine une ligne de `N` étoiles. Le paramètre `N` est modifié à l'intérieur de la fonction mais pas à l'extérieur :

```
#include <stdio.h>

void ETOILES(int N) {
    while (N>0) {
       printf("*");
       N--;
    }
    printf("\n");
}

int main() {
    int compteur=14;
    ETOILES(compteur);
    printf("Valeur de compteur: %i\n",compteur);
    return 0;
}
```

L'appel de ce programme renvoit :

```
**************
Valeur de compteur: 14
```

la valeur de `compteur` n'a donc pas été modifié par l'appel de la fonction `ETOILES`.

5.6.3 Passage des paramètres par adresse

Comme on vient de le voir, tout paramètre est passé par valeur, et cette règle ne souffre *aucune* exception. Cela pose le problème de réaliser un passage de paramètre par adresse lorsque le programmeur en a besoin.
Pour changer la valeur d'une variable de la fonction appelante, on procède comme suit :
– la fonction appelante doit fournir l'adresse de la variable ;

– la fonction appelée doit déclarer le paramètre comme pointeur.
On peut alors atteindre la variable à l'aide du pointeur.
Exemple : Supposons qu'on désire écrire une procédure `add`, admettant trois paramètres `a`, `b` et `c`. On désire que le résultat de l'exécution de `add` soit d'affecter au paramètre `c` la somme des valeurs des deux premiers paramètres.
Le paramètre `c` ne peut évidemment pas être passé par valeur, puisqu'on désire modifier la valeur du paramètre effectif correspondant. Il faut donc programmer `add` de la manière suivante :

```
void add(int a, int b, int *c) {
/* c repère l'entier où on veut mettre le résultat   */
   *c = a + b;
}
int main() {
  int i=10,j=14,k;
  /* on passe les valeurs de i et j comme premiers paramètres */
  /* on passe l'adresse de k comme troisième paramètre        */
  add(i,j,&k);
}
```

5.6.4 Passage de tableau en paramètre

Comme il est impossible de passer 'la valeur' de tout un tableau à une fonction, on fournit l'adresse d'un élément du tableau[4].
En général, on fournit l'adresse du premier élément du tableau, qui est donnée par le nom du tableau.
Dans la liste des paramètres d'une fonction, on peut déclarer un tableau par le nom suivi de crochets,

```
<type> <nom>[]
```

ou simplement par un pointeur sur le type des éléments du tableau :

```
<type> *<nom>
```

On préfèrera la première écriture car il n'est pas possible dans la seconde de savoir si le programmeur a voulu passer en paramètre un pointeur vers `<type>` (c'est à dire un pointeur vers un seul `<type>`), ou au contraire si il a voulu passer un tableau, c'est à dire un pointeur vers une zone de n `<type>`.
Exemple :

```
#include <stdlib.h>

/* Initialise les éléments d'un tableau */
void init_tab (int tab[], int n){
  int i;
  for (i = 0; i < n; i++)
     tab[i] = i;
  return;
}

int main() {
```

[4]Rappelons qu'un tableau est un pointeur constant (sur le premier élément du tableau)

```
  int i, n = 5;
  int *tab;
  tab = (int*)malloc(n * sizeof(int));
  init(tab,n);
}
```

Remarque : Quand une fonction admet un paramètre de type tableau, il y à deux cas possibles :

1. soit les différents tableaux qui lui sont passés en paramètre effectif ont des tailles différentes, et dans ce cas la taille doit être un paramètre supplémentaire de la fonction, comme dans l'exemple précédent ;
2. soit les différents tableaux qui lui sont passés en paramètre effectif ont tous la même taille, et dans ce cas la taille peut apparaître dans le type du paramètre effectif :

   ```
   #define NB_ELEM 10
   void init_tab (int tab[NB_ELEM]){{
      ...
   }
   ```

5.7 Fonction à nombre variable de paramètres

Il est possible de déclarer une fonction comme ayant un nombre variable de paramètres en "déclarant" les paramètres optionnels par l'unité lexicale "`...`" (3 points à la suite). Une fonction peut avoir à la fois des paramètres obligatoires et des paramètres optionnels, les paramètres obligatoires apparaissant en premier et l'unité lexicale "`...`" apparaissant en dernière position dans la liste de déclaration des paramètres formels.
Dans le corps de la fonction on ne dispose pas de nom pour désigner les paramètres. L'accès à ceux-ci ne peut se faire qu'en utilisant les macros suivantes définies dans la bibliothèque standard `<stdarg.h>` :

va_list permet de déclarer une variable opaque au programmeur, à passer en paramètre aux autres macros. Cette variable s'appelle traditionnellement `ap` (pour argument pointer), et a pour but de repérer le paramètre effectif courant.

va_start doit être appelée avant toute utilisation de `va_arg`. La macro `va_start` a deux paramètres : la variable `ap` et le nom du dernier paramètre obligatoire de la fonction.

va_arg délivre le paramètre effectif courant : le premier appel à `va_arg` délivre le premier paramètre, puis chaque nouvel appel à `va_arg` délivre le paramètre suivant. La macro `va_arg` admet deux paramètres : la variable `ap` et le type du paramètre courant.

va_end doit être appelée après toutes les utilisations de `va_arg`. La macro `va_end` admet un seul paramètre : la variable `ap`.

Rien n'est prévu pour communiquer à la fonction le nombre et le type des paramètres effectivement passés : c'est un problème à la charge du programmeur.

Exemple Voici la définition d'une fonction de débuggage sur le modèle de `printf` admettant un nombre variable d'arguments. L'affichage ne s'effectue que si le niveau spécifié à l'appel (par le parametre `level`) est inférieur à une variable globale `DEBUG_LEVEL`.

```
#define DEBUG_LEVEL
...
/**
 * Fonction de debuggage
 */
int trace (int level, char * chaine, ...) {
   va_list args;//les arguments du println
   int length = 0;
   va_start(args, chaine);//initialisation de la structure va_list a partir
                          //d'un parametre d'appel de la fonction
   if (level <= DEBUG_LEVEL) {
     while (chaine[length]!=0) {
       //traitement des argument %...
       if (chaine[length] == '%') {
         length++;//on passe au caractere suivant
         switch (chaine[length]) {
         case 'd':
           printf("%d", va_arg(args, long));
           break;
         case 'i':
           printf("%i", va_arg(args, long));
           break;
         case 'x':
           printf("%x", va_arg(args, unsigned long));
           break;
         case 'X':
           printf("%X", va_arg(args, unsigned long));
           break;
         case 's':
           printf("%s", va_arg(args, char *));
           break;
         }
       } else {
         //on a une chaine classique donc on l'affiche
         printf("%c", chaine[length]);
       }
       length++;
     }
     //on termine correctement l'utilisation d'une structure va_list
     va_end(args);
     printf("\n");//on va a la ligne!
     return 0;
   }
}
```

Une utilisation classique d'une telle fonction, on associera à l'affichage d'informations de debuggage précises un niveau `level` élevé tandis que les affichages de base auront un paramètre `level` faible. Ensuite, c'est l'utilisateur qui en

faisant varier la valeur de `DEBUG_LEVEL` modifie la quantité et la précision des informations affichées à l'exécution. Exemple :

```
/**
 *  Insertion d'une valeur val dans un tableau tab de taille n
 *  contenant n-1 premiers elements tries
 */
void insertion(int tab[], int n, int val) {
    int i=n;
    trace(1, "Debut de insertion - param n=%i, val=%i",n,val);
    trace(2, " Debut decalage des elements");
    while( i!=0 && val < tab[i-1]) {
        trace(3, "Decalage de tab[%i] en position %i",i-1,i);
        tab[i] = tab[i-1];
        i--;
    }
    trace(2, " Fin decalage des elements");
    trace(2, " Insertion de la valeur %i en position %i",val,i)
    tab[i] = val;
    trace(1, "Fin de insertion");
}
```

A l'appel de cette fonction, on aura le comportement suivant :
- si `DEBUG_LEVEL = 0` : pas d'affichage ;
- si `DEBUG_LEVEL = 1` : l'entrée et la sortie de la fonction sera visible par un message affiché. Cela permet de savoir rapidement en cas de bug quelle fonction pose problème ;
- si `DEBUG_LEVEL = 2`, on aura en plus l'affichage des étapes dans l'algorithme utilisé pour la fonction ;
- si `DEBUG_LEVEL > 2`, on aura tous les détails possibles.

5.8 Pointeurs sur une fonction

Il est parfois utile de passer une fonction comme paramètre d'une autre fonction. Cette procédure permet en particulier d'utiliser une même fonction pour différents usages. Pour cela, on utilise un mécanisme de pointeur.
Un pointeur sur une fonction correspond à l'adresse du début du code de la fonction. Un pointeur `p_fonction` sur une fonction ayant pour prototype
`type fonction(type_1 arg_1,...,type_n arg_n);`
se déclare par :
`type (*p_fonction)(type_1,...,type_n);`
Ainsi, considérons une fonction `operateur_binaire` prenant pour paramètres deux entiers `a` et `b`, ainsi qu'une fonction `f` de type `int` qui prend elle-même deux entiers en paramètres. On suppose que `operateur_binaire` renvoit l'entier `f(a,b)`.
Le prototype de cette fonction sera :
`int operateur_binaire(int a, int b, int (*f)(int, int))`
Pour appeler la fonction `operateur_binaire`, on utilisera comme troisième paramètre effectif l'identificateur de la fonction utilisée. Par exemple, si `somme` est la fonction

`int somme(int a, int b) { return a+b; }`
on appelle la fonction `operateur_binaire` pour la fonction `somme` par l'expression :
`operateur_binaire(a,b,somme)`
A noter que la notation `&somme` **n'est pas** utilisée comme paramètre effectif.
Pour appeler la fonction passée en paramètre dans le corps de la fonction `operateur_binaire`, on écrit `(*f)(a, b)`. Ainsi, cette fonction pourrait s'écrire :

```
int operateur_binaire(int a, int b, int (*f)(int, int)) {
  return((*f)(a,b));
}
```

Les pointeurs sur les fonctions sont notamment utilisés dans la fonction de tri des éléments d'un tableau `qsort` et dans la recherche d'un élément dans un tableau `bsearch`. Ces deux fonctions sont définies dans la librairie standard `<stdlib.h>`.
Le prototype de la fonction de tri[5] (algorithme quicksort) est ainsi :

```
void  qsort(void *tab, size_t nb_elements, size_t taille_elements,
            int(*compar)(const void *, const void *));
```

Cette gymnastique sur les pointeurs de fonction n'est pas forcément triviale mais reste utile dans certains cas, en particulier dans la définition de fonction "générique" comme la fonction `qsort` évoquée précédemment.

[5] Elle permet de trier les `nb_elements` premiers éléments du tableau `tab`. Le paramètre `taille_elements` donne la taille des éléments du tableau. Le type `size_t` utilisé ici est un type prédéfini dans `<stddef.h>`. Il correspond au type du résultat de l'évaluation de `sizeof`.
La fonction `qsort` est paramétrée par la fonction de comparaison `compar` de prototype :
`int compar(void *a, void *b);`
Les deux paramètres `a` et `b` de la fonction `compar` sont des pointeurs génériques de type `void *`. Ils correspondent à des adresses d'objets dont le type n'est pas déterminé. Cette fonction de comparaison retourne

- un entier qui vaut 0 si les deux objets pointés par a et b sont égaux ;
- une valeur strictement négative (resp. positive) si l'objet pointé par `a` est strictement inférieur (resp. supérieur) à celui pointé par `b`.

Chapitre 6

Gestion des fichiers

Comme beaucoup de langages, le C offre la possibilité de lire et d'écrire des données dans un fichier.
Pour des raisons d'efficacité, les accès à un fichier se font par l'intermédiaire d'une mémoire-tampon (on parle de buffer), ce qui permet de réduire le nombre d'accès aux périphériques (disque...).
Pour pouvoir manipuler un fichier, un programme a besoin d'un certain nombre d'informations : l'adresse de l'endroit de la mémoire-tampon où se trouve le fichier, la position de la tête de lecture, le mode d'accès au fichier (lecture ou écriture)... Ces informations sont rassemblées dans une structure, dont le type, `FILE *`, est défini dans `<stdio.h>`. Un objet de type `FILE *` est appelé *flot de données* (*stream* en anglais).
Avant de lire ou d'écrire dans un fichier, on notifie son accès par la commande `fopen`. Cette fonction prend comme argument le nom du fichier, défini avec le système d'exploitation le mode d'ouverture du fichier et initialise un flot de données, qui sera ensuite utilisé lors de l'écriture ou de la lecture. Après les traitements, on annule la liaison entre le fichier et le flot de données grâce à la fonction `fclose`.

6.1 Ouverture et fermeture de fichiers

6.1.1 Ouverture de fichiers : la fonction `fopen`

Lorsqu'on désire accéder à un fichier, il est nécessaire avant tout accès d'ouvrir le fichier à l'aide de la fonction `fopen`.
Cette fonction, de type `FILE *` ouvre un fichier et lui associe un flot de données. Sa syntaxe est :

```
fopen("nom-de-fichier","mode")
```

La valeur retournée par `fopen` est
– soit un flot de données ;
– soit `NULL` si l'exécution de cette fonction ne se déroule pas normalement.
Je vous conseille donc de **toujours** tester si la valeur renvoyée par la fonction `fopen` est égale à `NULL` afin de détecter les erreurs d'ouverture de fichier (lecture d'un fichier inexistant...).

Le premier argument de `fopen` fournit donc le nom du fichier. Le second argument, `mode`, est une chaîne de caractères qui spécifie le mode d'accès au fichier. Les spécificateurs de mode d'accès diffèrent suivant le type de fichier considéré. On distingue

– les fichiers *textes*, pour lesquels les caractères de contrôle (retour à la ligne ...) seront interprétés en tant que tels lors de la lecture et de l'écriture ;
– les fichiers *binaires*, pour lesquels les caractères de contrôle se sont pas interprétés.

Les différents modes d'accès sont listés dans le tableau 6.1.

`"r"`	ouverture d'un fichier texte en lecture
`"w"`	ouverture d'un fichier texte en écriture
`"a"`	ouverture d'un fichier texte en écriture à la fin
`"rb"`	ouverture d'un fichier binaire en lecture
`"wb"`	ouverture d'un fichier binaire en écriture
`"ab"`	ouverture d'un fichier binaire en écriture à la fin
`"r+"`	ouverture d'un fichier texte en lecture/écriture
`"w+"`	ouverture d'un fichier texte en lecture/écriture
`"a+"`	ouverture d'un fichier texte en lecture/écriture à la fin
`"r+b"`	ouverture d'un fichier binaire en lecture/écriture
`"w+b"`	ouverture d'un fichier binaire en lecture/écriture
`"a+b"`	ouverture d'un fichier binaire en lecture/écriture à la fin

TAB. 6.1 – Les modes d'accès aux fichiers dans la fonction `fopen`

Conditions particulières et cas d'erreur

– Si le mode contient la lettre `r`, le fichier doit exister, sinon c'est une erreur.
– Si le mode contient la lettre `w`, le fichier peut ne pas exister. Dans ce cas, il sera créé, et si le fichier existait déjà, son ancien contenu est perdu.
– Si le mode contient la lettre `a`, le fichier peut ne pas exister. Comme pour le cas précédent, si le fichier n'existe pas, il est créé ; si le fichier existe déjà, son ancien contenu est conservé.
– Si un fichier est ouvert en mode "écriture à la fin", toutes les écritures se font à l'endroit qui est était la fin du fichier lors de l'exécution de l'ordre d'écriture. Cela signifie que si plusieurs processus partagent le même `FILE*`, résultat de l'ouverture d'un fichier en écriture à la fin, leurs écritures ne s'écraseront pas mutuellement.

Les modes de communication standard

Quand un programme est lancé par le système, celui-ci ouvre trois fichiers correspondant aux trois modes de communication standard : *standard input*, *standard output* et *standard error*. Il y a trois constantes prédéfinies dans `stdio.h` qui correspondent aux trois fichiers :

– `stdin` (standard input) : flot d'entrée (par défaut, le clavier) ;

– `stdout` (standard output) : flot de sortie (par défaut, l'écran) ;
– `stderr` (standard error) : flot d'affichage des messages d'erreur (par défaut, l'écran).

Il est fortement conseillé d'afficher systématiquement les messages d'erreur sur `stderr` afin que ces messages apparaissent à l'écran même lorsque la sortie standard est redirigée.

Utilisation typique de `fopen`

```
#include <stdio.h>
FILE *fp;
...
if ((fp = fopen("donnees.txt","r")) == NULL) {
   fprintf(stderr,"Impossible d'ouvrir le fichier données en lecture\n");
   exit(1);
}
```

6.1.2 Fermeture de fichiers : la fonction `fclose`

Elle permet de fermer le flot qui a été associé à un fichier par la fonction `fopen`. Sa syntaxe est :

`fclose(flot)`

où `flot` est le flot de type `FILE*` retourné par la fonction `fopen` correspondante. La fonction `fclose` retourne 0 si l'opération s'est déroulée normalement et `EOF` si il y a eu une erreur.

6.2 Les entrées-sorties formatées

6.2.1 La fonction d'écriture en fichier fprintf

La fonction `fprintf`, analogue à `printf` (voir §2.5.4 page 29), permet d'écrire des données dans un flot. Sa syntaxe est :

`fprintf(flot,"chaîne de contrôle"`, $expression_1$, ..., $expression_n$`);`

où `flot` est le flot de données retourné par la fonction `fopen`. Les spécifications de format utilisées pour la fonction `fprintf` sont les mêmes que pour `printf` puisque :

`printf(...)` $\iff$ `fprintf(stdout,...)`

6.2.2 La fonction de saisie en fichier `fscanf`

La fonction `fscanf`, analogue à `scanf` (voir §2.5.5 page 31), permet de lire des données dans un fichier. Sa syntaxe est semblable à celle de `scanf` :

`fscanf(flot,"chaîne de contrôle"`, arg_1, ..., arg_n`);`

où `flot` est le flot de données retourné par `fopen`. Les spécifications de format sont ici les mêmes que celles de la fonction `scanf`.

6.3 Impression et lecture de caractères dans un fichier

6.3.1 Lecture et écriture par caractère : `fgetc` et `fputc`

Similaires aux fonctions `getchar` et `putchar` (voir §2.5.1 et §2.5.2), les fonctions `fgetc` et `fputc` permettent respectivement de lire et d'écrire un caractère dans un fichier.
La fonction `fgetc` retourne le caractère lu dans le fichier et la constante `EOF` lorsqu'elle détecte la fin du fichier. Son prototype est :

```
int fgetc(FILE* flot);
```

où `flot` est le flot de type `FILE*` retourné par la fonction `fopen`.
La fonction `fputc` écrit un caractère dans le flot de données :

```
int fputc(int caractere, FILE *flot)
```

Elle retourne l'entier correspondant au caractère lu (ou la constante `EOF` en cas d'erreur).

6.3.2 Lecture et écriture optimisées par caractère : `getc` et `putc`

Il existe également deux versions optimisées des fonctions `fgetc` et `fputc` qui sont implémentées par des macros. Il s'agit respectivement de `getc` et `putc`. Leur syntaxe est similaire à celle de `fgetc` et `fputc` :

```
int getc(FILE* flot);
int putc(int caractere, FILE *flot)
```

Ainsi, le programme suivant lit le contenu du fichier texte `entree.txt`, et le recopie caractère par caractère dans le fichier `sortie.txt` :

```
#include <stdio.h>
#include <stdlib.h>
#define ENTREE "entree.txt"
#define SORTIE "sortie.txt"

int main(void) {
    FILE *f_in, *f_out;
    int c;
    // Ouverture du fichier ENTREE en lecture
    if ((f_in = fopen(ENTREE,"r")) == NULL) {
        fprintf(stderr, "\nErreur: Impossible de lire %s\n",ENTREE);
        return(EXIT_FAILURE);
    }
    // Ouverture du fichier SORTIE en ecriture
    if ((f_out = fopen(SORTIE,"w")) == NULL) {
        fprintf(stderr, "\nErreur: Impossible d'ecrire dans %s\n",SORTIE);
        return(EXIT_FAILURE);
    }
```

```
    // Recopie du contenu de ENTREE dans SORTIE
    while ((c = fgetc(f_in)) != EOF)
        fputc(c, f_out);
    // Fermeture des flots de donnees
    fclose(f_in);
    fclose(f_out);
    return(EXIT_SUCCESS);
}
```

On notera l'utilisation des constantes `EXIT_SUCCESS` (valant 0) et `EXIT_FAILURE` (valant 1), définies dans la librairie `<stdlib.h>` et qui sont les paramètres privilégiés de la fonction `exit` (voir §9.19).

6.3.3 Relecture d'un caractère

Il est possible de replacer un caractère dans un flot au moyen de la fonction `ungetc` :

```
int ungetc(int carac, FILE *f);
```

Cette fonction place le caractère `carac` (converti en `unsigned char`) dans le flot `f`. En particulier, si `carac` est égal au dernier caractère lu dans le flot, elle annule le déplacement provoqué par la lecture précédente. Toutefois, `ungetc` peut être utilisée avec n'importe quel caractère (sauf `EOF`). L'exemple suivant permet d'illustrer le comportement de `ungetc` :

```
#include <stdio.h>
#include <stdlib.h>
#define ENTREE "entree.txt"

int main(void) {
    FILE *f_in;
    int c;
    if ((f_in = fopen(ENTREE,"r")) == NULL) {
      fprintf(stderr, "\nErreur: Impossible de lire le fichier %s\n",ENTREE);
      return(EXIT_FAILURE);
    }
    while ((c = fgetc(f_in)) != EOF) {
      if (c == '0')
        ungetc('.',f_in);
      putchar(c);
    }
    fclose(f_in);
    return(EXIT_SUCCESS);
}
```

Sur le fichier `entree.txt` dont le contenu est `097023`, ce programme affiche à l'écran `0.970.23`.

6.3.4 Les entrées-sorties binaires : `fread` et `fwrite`

Les fonctions d'entrées-sorties binaires permettent de transférer des données dans un fichier sans transcodage. En ce sens, elles sont plus portables car elles écrivent ce qu'on leur dit. Elles sont ainsi plus efficaces que les fonctions d'entrée-sortie standard.
Elles sont notamment utiles pour manipuler des données de grande taille ou ayant un type composé. Leurs prototypes sont :

```
size_t fread(void *pointeur, size_t taille, size_t nbre, FILE *f);
size_t fwrite(void *pointeur, size_t taille, size_t nbre, FILE *f);
```

où `pointeur` est l'adresse du début des données à transférer, `taille` la taille des objets à transférer et `nbre` leur nombre. Rappelons que le type `size_t`, défini dans `<stddef.h>`, correspond au type du résultat de l'évaluation de `sizeof` (voir §2.1.7 et §2.4.4).
La fonction `fread` lit les données sur le flot `f` et la fonction `fwrite` les écrit. Ces deux fonctions retournent le nombre de données transférées.
Par exemple, le programme suivant écrit un tableau d'entiers (contenant les 50 premiers entiers) avec `fwrite` dans le fichier `sortie.txt`, puis lit ce fichier avec `fread` et imprime les éléments du tableau.

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

#define NB 50
#define F_SORTIE "sortie.txt"
int main(void) {
    FILE *f_in, *f_out;
    int *tab1, *tab2;
    int i;
    // allocation memoire des tableaux
    tab1 = (int*)malloc(NB * sizeof(int));
    tab2 = (int*)malloc(NB * sizeof(int));
    for (i = 0 ; i < NB; i++)
       tab1[i] = i;

    /* ecriture du tableau dans F_SORTIE */
    if ((f_out = fopen(F_SORTIE, "w")) == NULL) {
       fprintf(stderr, "\nImpossible d'ecrire dans %s\n",F_SORTIE);
       return(EXIT_FAILURE);
    }
    fwrite(tab1, NB * sizeof(int), 1, f_out);
    fclose(f_out);

    /* lecture dans F_SORTIE */
    if ((f_in = fopen(F_SORTIE, "r")) == NULL) {
       fprintf(stderr, "\nImpossible de lire dans %s\n",F_SORTIE);
       return(EXIT_FAILURE);
    }
    fread(tab2, NB * sizeof(int), 1, f_in);
```

```
    fclose(f_in);
    for (i = 0 ; i < NB; i++)
       printf("%d\t",tab2[i]);
    printf("\n");
    return(EXIT_SUCCESS);
}
```

Les éléments du tableau sont bien affichés à l'écran. Par contre, on constate que le contenu du fichier sortie n'est pas encodé.

6.3.5 Positionnement dans un fichier : `fseek`, `rewind` et `ftell`

Les différentes fonctions d'entrées-sorties permettent d'accéder à un fichier en mode séquentiel : les données du fichier sont lues ou écrites les unes à la suite des autres.
Il est également possible d'accéder à un fichier en mode direct, c'est-à-dire que l'on peut se positionner à n'importe quel endroit du fichier. La fonction `fseek` permet de se positionner à un endroit précis et a pour prototype :

```
int fseek(FILE *flot, long deplacement, int origine);
```

La variable `deplacement` détermine la nouvelle position dans le fichier. Il s'agit d'un déplacement relatif par rapport à `origine`, compté en nombre d'octets. La variable `origine` peut prendre trois valeurs :

1. `SEEK_SET` (égale à 0) : début du fichier ;
2. `SEEK_CUR` (égale à 1) : position courante ;
3. `SEEK_END` (égale à 2) : fin du fichier.

La fonction
`int rewind(FILE *flot);`

permet de se positionner au début du fichier. Elle est équivalente à `fseek(flot, 0, SEEK_SET)` ;
La fonction
`long ftell(FILE *flot);`

retourne la position courante dans le fichier (en nombre d'octets depuis l'origine). L'exemple suivant illustre l'utilisation des fonctions précédentes :

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

#define NB 50
#define F_SORTIE "sortie.txt"
int main(void) {
    FILE *f_in, *f_out;
    int *tab;
    int i;
    // Initialisation du tableau
```

```
    tab = (int*)malloc(NB * sizeof(int));
    for (i = 0 ; i < NB; i++)
      tab[i] = i;
    /* ecriture du tableau dans F_SORTIE */
    if ((f_out = fopen(F_SORTIE, "w")) == NULL){
        fprintf(stderr, "\nImpossible d'ecrire dans %s\n",F_SORTIE);
        return(EXIT_FAILURE);
    }
    fwrite(tab, NB * sizeof(int), 1, f_out);
    fclose(f_out);
    /* lecture dans F_SORTIE */
    if ((f_in = fopen(F_SORTIE, "r")) == NULL) {
        fprintf(stderr, "\nImpossible de lire dans %s\n",F_SORTIE);
        return(EXIT_FAILURE);
    }
    /* on se positionne a la fin du fichier */
    fseek(f_in, 0, SEEK_END);
    printf("\n position %ld", ftell(f_in));
    /* deplacement de 10 int en arriere */
    fseek(f_in, -10 * sizeof(int), SEEK_END);
    printf("\n position %ld", ftell(f_in));
    fread(&i, sizeof(i), 1, f_in);
    printf("\t i = %d", i);
    /* retour au debut du fichier */
    rewind(f_in);
    printf("\n position %ld", ftell(f_in));
    fread(&i, sizeof(i), 1, f_in);
    printf("\t i = %d", i);
    /* deplacement de 5 int en avant */
    fseek(f_in, 5 * sizeof(int), SEEK_CUR);
    printf("\n position %ld", ftell(f_in));
    fread(&i, sizeof(i), 1, f_in);
    printf("\t i = %d\n", i);
    fclose(f_in);
    return(EXIT_SUCCESS);
}
```

L'exécution de ce programme affiche à l'écran :

```
 position 200
 position 160    i = 40
 position 0      i = 0
 position 24     i = 6
```

On constate en particulier que l'emploi de la fonction `fread` provoque un déplacement correspondant à la taille de l'objet lu à partir de la position courante.

Chapitre 7

Les directives du préprocesseur

Le préprocesseur est un programme exécuté lors de la première phase de la compilation. Il effectue des modifications textuelles sur le fichier source à partir de directives. Les différentes directives au préprocesseur, introduites par le caractère `#`, ont pour but :
– l'incorporation de fichiers source (`#include`),
– la définition de constantes symboliques et de macros (`#define`),
– la compilation conditionnelle (`#if`, `#ifdef`,...).
L'utilisation de ces directives a déjà été introduite au §1.4 page 7. Il s'agit ici de détailler l'ensemble des directives du préprocesseur.

7.1 La directive `#include`

Elle permet d'incorporer dans le fichier source le texte figurant dans un autre fichier. Ce dernier peut être un fichier en-tête de la librairie standard (`stdio.h`, `math.h`,...) ou n'importe quel autre fichier. La directive `#include` possède deux syntaxes voisines :
`#include <nom-de-fichier>`
recherche le fichier mentionné dans un ou plusieurs répertoires systèmes définis par l'implémentation (typiquement `/usr/include/`) :
`#include "nom-de-fichier"`
recherche le fichier dans le répertoire courant (celui où se trouve le fichier source). On peut spécifier d'autres répertoires à l'aide de l'option `-I` du compilateur (voir chapitre 8).
La première syntaxe est généralement utilisée pour les fichiers en-tête de la librairie standard, tandis que la seconde est plutôt destinée aux fichiers créés par l'utilisateur.

7.2 La directive `#define`

La directive `#define` permet de définir des constantes symboliques (on parle aussi de macros sans paramètres) ou des macros avec paramètre.

7.2.1 Définition de constantes symboliques

Bien que cela a déjà été vu précédemment, rappelons que lorsque le préprocesseur lit une ligne du type :
`#define nom reste-de-la-ligne`

il remplace dans toute la suite du source toute nouvelle occurrence de `nom` par `reste-de-la-ligne`. Il n'y a aucune contrainte quand à ce qui peut se trouver dans `reste-de-la-ligne` : on pourra ainsi écrire

```
#define BEGIN {
#define END }
```

L'utilité principale des macros sans paramètre est de donner un nom parlant à une constante. Les avantages à toujours donner un nom aux constantes sont les suivants :

1. un nom bien choisi permet d'expliciter la sémantique de la constante. Ex : `#define NB_COLONNES 100`.
2. la constante chiffrée se trouve à un seul endroit, ce qui facilite la modification du programme quand on veut changer la valeur de la constante (cas de la taille d'un tableau, par exemple).
3. on peut expliciter facilement les relations entre constantes. Ex :

   ```
   #define NB_LIGNES 24
   #define NB_COLONNES 80
   #define TAILLE_MATRICE NB_LIGNES * NB_COLONNES
   ```

Définition de constantes symboliques à l'invocation du compilateur

Certains compilateurs permettent de définir des constantes symboliques à l'invocation du compilateur. Il est alors possible d'écrire un programme utilisant une macro qui n'est nulle part définie dans le source.
Ceci est très pratique pour que certaines constantes critiques d'un programme aient une valeur qui soit attribuée à l'extérieur du programme (comme lors d'une une phase de configuration par exemple).
Ci-dessous, un exemple pour `gcc` : la compilation du fichier `prog.c` en définissant la constante symbolique `NB_LIGNES` de valeur 14 :
`gcc -c -DNB_LIGNES=24 prog.c`

Constantes symboliques prédéfinies

Il y a un certain nombre de macros prédéfinies par le préprocesseur, récapitulées dans le tableau 7.1.

7.2.2 Les macros avec paramètres

Une macro avec paramètres se définit de la manière suivante :
`#define nom(liste-de-paramètres) corps-de-la-macro`

Nom	Valeur de la macro	Type
`__LINE__`	numéro de la ligne courante du programme source	entier
`__FILE__`	nom du fichier source en cours de compilation	chaîne
`__DATE__`	la date de la compilation	chaîne
`__TIME__`	l'heure de la compilation	chaîne
`__STDC__`	1 si le compilateur est ISO, 0 sinon	entier

TAB. 7.1 – Les constantes symboliques prédéfinies

où `liste-de-paramètres` est une liste d'identificateurs séparés par des virgules. Par exemple, avec la directive
`#define MAX(a,b) (a > b ? a : b)`
le processeur remplacera dans la suite du code toutes les occurences du type `MAX(x,y)` où x et y sont des symboles quelconques par `(x > y ? x : y)`
Une macro a donc une syntaxe similaire à celle d'une fonction, mais son emploi permet en général d'obtenir de meilleures performances en temps d'exécution.
La distinction entre une définition de constante symbolique et celle d'une macro avec paramètres se fait sur le caractère qui suit immédiatement le nom de la macro : si ce caractère est une parenthèse ouvrante, c'est une macro avec paramètres, sinon c'est une constante symbolique. Il ne faut donc jamais mettre d'espace entre le nom de la macro et la parenthèse ouvrante. L'erreur classique étant d'écrire par erreur :
`#define CARRE (a) a * a`
la chaîne de caractères `CARRE(2)` sera alors remplacée par : `(a) a * a (2)`
Il faut toujours garder à l'esprit que le préprocesseur n'effectue que des remplacements de chaînes de caractères. En particulier, il est conseillé de toujours mettre entre parenthèses le corps de la macro et les paramètres formels qui y sont utilisés. Par exemple, si l'on écrit sans parenthèses :
`#define CARRE(a) a * a`
le préprocesseur remplacera `CARRE(a + b)` par `a + b * a + b` et non par `(a + b) * (a + b)`. De même, `!CARRE(x)` sera remplacé par `! x * x` et non par `!(x * x)`.
Enfin, il faut être attentif aux éventuels effets de bord que peut entraîner l'usage de macros. Par exemple, `CARRE(x++)` aura pour expansion `(x++) * (x++)`. L'opérateur d'incrémentation sera donc appliqué deux fois au lieu d'une.
En conclusion, les macros avec paramètres sont à utiliser avec précaution et en cas de doutes, il faut mieux s'en passer.

7.3 La compilation conditionnelle

La compilation conditionnelle a pour but d'incorporer ou d'exclure des parties du code source dans le texte qui sera généré par le préprocesseur, le choix étant basé sur un test exécuté à la compilation. Elle permet d'adapter le programme au matériel ou à l'environnement sur lequel il s'exécute, ou d'introduire dans le programme des instructions de débuggage.

Les directives de compilation conditionnelle se répartissent en deux catégories, suivant le type de la condition invoquée qui est testée :
– la valeur d'une expression
– l'existence ou l'inexistence de symboles

7.3.1 Condition liée à la valeur d'une expression

Sa syntaxe la plus générale est :

```
#if condition-1
  partie-du-programme-1
#elif condition-2
  partie-du-programme-2
     ...
#elif condition-n
  partie-du-programme-n
#else
  partie-du-programme-else
#endif
```

Le nombre de `#elif` est quelconque et le `#else` est facultatif. Chaque `condition-i` doit être une expression constante.
Lors de la compilation, une seule `partie-du-programme-i` sera compilée : celle qui correspond à la première `condition-i` non nulle, ou bien la `partie-du-programme-else` si toutes les conditions sont nulles.
Par exemple, on peut écrire :

```
#define PROCESSEUR ALPHA
...
#if PROCESSEUR == ALPHA
    taille_long = 64;
#elif PROCESSEUR == PC
    taille_long = 32;
#endif
```

7.3.2 Condition liée à l'existence d'un symbole

Sa syntaxe est la suivante :

```
#ifdef symbole
  partie-du-programme-1
#else condition-2
  partie-du-programme-2
#endif
```

Si `symbole` est défini au moment où l'on rencontre la directive `#ifdef`, alors `partie-du-programme-1` sera compilée et `partie-du-programme-2` sera ignorée. Dans le cas contraire, c'est `partie-du-programme-2` qui sera compilée. La directive `#else` est comme précédemment facultative. Ce type de directive est utile pour rajouter des instructions destinées au débuggage du programme :

```
#define DEBUG
  ....
#ifdef DEBUG
  for (i = 0; i < N; i++)
    printf("%d\n",i);
#endif /* DEBUG */
```

Il suffit alors de supprimer la directive `#define DEBUG` pour que les instructions liées au debuggage ne soient pas compilées[1].
De façon similaire, on peut tester la non-existence d'un symbole à l'aide de la directive `#ifndef` :

```
#ifndef symbole
  partie-du-programme-1
#else condition-2
  partie-du-programme-2
#endif
```

On rencontre beaucoup cette directive dans le cadre de la compilation modulaire (voir chapitre 8) puisqu'elle permet de s'assurer que les définitions d'instructions dans un fichier d'en-tête ne seront effectuées qu'une seule fois.

7.3.3 L'opérateur `defined`

L'opérateur `defined` est un opérateur spécial : il ne peut être utilisé que dans le contexte d'une commande `#if` ou `#elif`. Il peut être utilisé sous l'une des deux formes suivantes :
– `defined nom`
– `defined ( nom )`
Il délivre la valeur 1 si `nom` est une macro définie, et la valeur 0 sinon. L'intérêt de cet opérateur est de permettre d'écrire des tests portant sur la définition de plusieurs macros, alors que `#ifdef` ne peut en tester qu'une :
`#if defined(SOLARIS) || defined(SYSV)`

7.3.4 La commande `#error`

La commande `#error` a la syntaxe suivante :
`#error chaine`
La rencontre de cette commande provoquera l'émission d'un message d'erreur comprenant la `chaine`. Cette commande a pour utilité de capturer à la compilation des conditions qui font que le programme ne peut pas s'exécuter sur cette plate-forme. Voici un exemple où on teste que la taille des entiers est suffisante :

```
#include <limits.h>
#if INT_MAX < 1000000
#error "Entiers trop petits sur cette machine"
#endif
```

[1] A noter aussi que comme on l'a vu précedemment, on peut remplacer cette dernière directive par l'option de compilation `-DDEBUG`, qui permet de définir ce symbole.

7.3.5 La commande `#pragma`

La directive `#pragma` est liée à une implémentation spécifique et permet de définir, pour un compilateur donné, une directive de prétraitement. Sa syntaxe est la suivante :

```
#pragma commande
```

Chapitre 8

La programmation modulaire

Dès que l'on écrit un programme de taille importante ou destiné à être utilisé et maintenu par d'autres personnes, il est indispensable de se fixer un certain nombre de règles d'écriture. En particulier, il est nécessaire de fractionner le programme en plusieurs fichiers sources, que l'on compile séparemment.
Ces règles d'écriture ont pour objectifs de rendre un programme lisible, portable, réutilisable mais surtout facile à maintenir et à modifier.

L'idée est alors de regrouper dans un même fichier les instructions implémentant des fonctionnalités similaires (par exemple, `tab_management.c` contiendra une bibliothèque de fonctions gérant les tableaux, `file_management.c` fournira une bibliothèque de fonctions gérant les fichiers dans le contexte du projet effectué et `main.c` contiendra la définition de la fonction `main`). La programmation modulaire consiste alors à opérer les manipulations nécessaires permettant de lier ces fichiers.

8.1 Principes élémentaires

Trois principes essentiels doivent guider l'écriture d'un programme C. Ces principes s'appliquent en fait dans le cas général du génie logiciel.

1. *L'abstraction des constantes littérales*
   L'utilisation explicite de constantes littérales dans le corps d'une fonction rend les modifications et la maintenance difficiles. Des instructions comme :
   ```
   fopen("mon_fichier", "r");
   perimetre = 2 * 3.14 * rayon;
   ```
   sont à proscrire (il faudrait définir des constantes fournissant le nom du fichier ou la valeur de π). Sauf cas très particuliers, les constantes doivent être définies comme des constantes symboliques au moyen de la directive `#define`.
2. *La factorisation du code*
   Le but est d'éviter les duplications de code. La présence d'une même portion de code à plusieurs endroits du programme est un obstacle à d'éven-

tuelles modifications. Les fonctions doivent donc être systématiquement utilisées pour éviter la duplication de code. Il ne faut pas craindre de définir une multitude de fonctions de petite taille.

3. *La fragmentation du code*

   Pour des raisons de lisibilité, il est pertinent de découper un programme en plusieurs fichiers. En plus, cette règle permet de réutiliser facilement une partie du code pour d'autres applications.

   On sépare alors le programme en modules, chaque module implémentant des fonctions sur un thème similaire et qui se traduiront physiquement par deux fichiers :

   (a) un fichier en-tête (on dit aussi de header) ayant l'extension `.h` et contenant le prototype des fonctions principales implémentées dans ce module.

   (b) un fichier source ayant l'extension `.c` contenant non seulement le corps des fonctions déclarées dans le fichier en-tête mais également celui des fonctions intermédiaires éventuellement utilisées. Ce fichier inclut évidemment le fichier en-tête par le biais de la directive `#include`.

   Une possibilité est de placer une partie du code dans un fichier en-tête (on dit aussi de header) ayant l'extension `.h` que l'on inclut dans le fichier contenant le programme principal à l'aide de la directive `#include`.

   L'exemple suivant illustre ce principe sur un programme qui saisit deux entiers au clavier et affiche leur produit. Sur cet exemple, il a été choisi de définir un module `arithmétique` qui fournit la fonction effectuant le produit[1]. Ce module est donc implémenté dans les fichiers `arithmétique.h` et `arithmétique.c`. Le programme est donc composé de trois fichiers : les 2 fichiers du module et le fichier principale contenant la fonction `main` appelé ici `main.c` :

```
/*********************************************/
/*  fichier: main.c                          */
/*  saisit 2 entiers et affiche leur produit */
/*********************************************/
#include <stdlib.h>
#include <stdio.h>
#include "arithmetique.h" // Le module d'arithmetique

int main(void) {
  int a, b, c;
  scanf("%d",&a);
  scanf("%d",&b);
  c = produit(a,b); // appel de la fonction définie dans arithmetique.h
  printf("\nle produit vaut %d\n",c);
  return EXIT_SUCCESS;
}

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

[1] Il s'agit évidemment d'un exemple très simpliste !

```
/*  fichier: arithmetique.h                                */
/*  Gère les opérations arithmétiques sur les entiers   */
/*  (Ici, seul le produit est implémenté                */
/*********************************************************/
int produit(int a, int b);

/*********************************************************/
/*  fichier: arithmetique.c                             */
/*  Corps des fonctions définies dans arithmetique.h */
/*********************************************************/
#include "arithmetique.h"
int produit(int a, int b) {
   return(a * b);
}
```

Remarquer que c'est exactement la procédure utilisée pour les fonctions de la librairie standard : les fichiers en-tête `.h` de la librairie standard sont constitués de déclarations de fonctions et de définitions de constantes symboliques (la seule information importante pour un utilisateur) et le corps des fonctions en lui-même est séparé.

8.2 Eviter les erreurs d'inclusions multiples

Une dernière règle, **très importante**, consiste à éviter les erreurs de double définition de fonctions qui se traduise à l'édition de lien par le message suivant (avec `gcc`) :

```
toto.o:toto.c multiple definition of '...'
toto.o: first defined here
```

Cette erreur se produit en général lors de l'inclusion multiple d'un même fichier en-tête.
Pour éviter cela, la méthode consiste à définir une constante à la première inclusion du fichier en-tête et d'ignorer le contenu de celui-ci si la constante a déjà été définie (donc si on a déjà fait une inclusion). Pour cela, on utilise la directive `#ifndef` (voir 7.3.2). Il est recommandé d'appeler la constante `__TOTO_H` pour le fichier `toto.h` En appliquant cette règle, le fichier `arithmetique.h` de l'exemple précédent devient :

```
/*********************************************************/
/*  fichier: arithmetique.h                              */
/*  Gère les opérations arithmétiques sur les entiers   */
/*  (Ici, seul le produit est implémenté                */
/*********************************************************/
#ifndef __ARITHMETIQUE_H
#define __ARITHMETIQUE_H
int produit(int a, int b);

#endif
```

Je vous recommande de toujours appliquer cette dernière règle dès lors que vous créez un fichier de header !

8.3 La compilation séparée

Ce n'est pas le tout de bien fragmenter son code en plusieurs fichiers, encore faut-il les compiler pour obtenir un executable.
La méthode consiste à générer un fichier objet par module (option `-c` de `gcc`) :

```
gcc -O3 -Wall -I. -c module_1.c
gcc -O3 -Wall -I. -c module_2.c
   ⋮
gcc -O3 -Wall -I. -c module_n.c
```

Ces commandes génèrent n fichiers objets `module_i.o`.
L'option `-I` de `gcc` permet d'ajouter un répertoire en première position de la liste des répertoires où sont cherchés les fichiers en-tête. (ici, le répertoire courant, `./` : l'option `-I` est donc en fait facultative dans ce cas). Cette option est utile lorsque, pour des raisons de lisibilité dans l'architecture des fichiers sources, un répertoire `Include` est créé pour contenir tous les fichiers en-tête du programme. La compilation avec `gcc` comportera alors l'option `-IInclude`.

Une passe d'édition de lien entre ces fichiers objets en ensuite nécessaire pour générer l'exécutable final `toto.exe` :

```
gcc -o toto.exe module_1.o module_2.o ... module_n.o
```

8.4 Résumé des règles de programmation modulaire

1. Découper le programme en modules implémentant des fonctions similaires.
2. Chaque module se traduit en 2 fichiers sources :
   - un fichier en-tête `module.h` qui définit son interface, c'est à dire le prototype des fonctions du module qui sont exportées. Plus précisément, ce fichier se compose :
     - des déclarations des fonctions d'interface (celles qui sont exportées et donc utilisées dans d'autres fichiers sources)
     - d'éventuelles définitions de constantes symboliques et de macros.
     - d'éventuelles directives au préprocesseur (inclusion d'autres fichiers, compilation conditionnelle).

     Ce fichier doit respecter également les conventions d'écritures proposées au §8.2 pour éviter les erreurs de définitions multiples.
   - un fichier `module.c` contenant le corps des fonctions implémentées dans ce module. Ce fichier se compose :
     - de variables globales qui ne sont utilisées que dans le fichier `module.c` ;
     - du corps des fonctions d'interface dont la déclaration se trouve dans `module.h` ;
     - d'éventuelles fonctions locales à `module.c`.
3. Le fichier `module.h` est inclus dans le fichier `module.c` (via la directive `#include`) et dans tous les autres fichiers qui font appel aux fonctions exportées par ce module.
4. Chaque module est compilé pour générer l'exécutable final (voir 8.3).

Ces règles s'ajoutent aux règles générales d'écriture de programmes C abordées au §1.6 page 9.

8.5 L'outils 'make'

Losrqu'un programme est fragmenté en plusieurs fichiers sources compilés séparemment, la procédure de compilation peut très vite devenir longue et fastidieuse.
Il est alors extrèmement pratique de l'automatiser à l'aide de l'utilitaire 'make' d'Unix. Une bonne utilisation de `make` permet de réduire le temps de compilation et également de garantir que celle-ci est effectuée correctement.

8.5.1 Principe de base

L'idée principale de make est d'effectuer uniquement les étapes de compilation nécessaires à la création d'un exécutable. Par exemple, si un seul fichier source a été modifié dans un programme composé de plusieurs fichiers, il suffit de recompiler ce fichier et d'effectuer l'édition de liens. Les autres fichiers sources n'ont pas besoin d'être recompilés.
La commande `make` recherche par défaut dans le répertoire courant un fichier `makefile` ou `Makefile`. Ce fichier spécifie les dépendances entre les différents fichiers sources, objets et exécutables.
Voyons tout de suite les principales options de cette commande (les plus utiles ; pour plus de détails : `man make`)
– `make -f <nom_fichier>` : utiliser le fichier `<nom_fichier>` à la place du traditionnel fichier `Makefile`.
– `make -C <path>` : exécuter le fichier `Makefile` situé dans le répertoire `<path>`.

8.5.2 Création d'un `Makefile`

Un fichier `Makefile` est composé d'une liste de règles de dépendances entre fichiers, de la forme :

```
cible: liste_de_dépendances
 <TAB> commandes_a_effectuer
```

La première ligne spécifie un fichier cible (généré par `commandes_a_effectuer`) et la liste des fichiers dont il dépend (séparés par des espaces) : il s'agit donc de la liste des ficiers requis pour la génération de `cible`.
Les lignes suivantes, qui **doivent** commencer par une tabulation (le caractère TAB), indiquent les commandes à exécuter pour générer le fichier `cible`. Ces commandes ne seront exécutées que si le fichier `cible` n'exécute pas ou si l'un des fichiers de dépendance est plus récent que le fichier `cible`.
De façon générale, l'écriture d'un fichier `Makefile` se rapproche largement de l'écriture d'un script shell :
– On peut placer des commentaires, qui commencent par `#` :
  `#Ceci est un commentaire`
  Les commentaires s'étendent sur une seule ligne (équivalent du `//` en C99).

– On peut declarer des variables.
  Une variable se déclare de la façon suivante : `SRC = machin.c`
  Ensuite, le contenu de cette variable (`machin.c`) est obtenu par l'appel `$(SRC)`

Ainsi, considérons l'exemple du programme effectuant le produit de deux entiers présenté au §**??**. On souhaite générer l'exécutable `toto`.

– Pour une compilation "à la main" et en suivant les recommendations du §1.1.4, il faudrait exécuter les commandes :
  ```
  gcc -g3 -Wall -c produit.c
  gcc -g3 -Wall -c main.c
  gcc -g3 -Wall -o toto main.o produit.o
  ```
  On l'aura compris : cela peut devenir très vite long et lassant !
– Un premier fichier `Makefile` gérant cette compilation pourrait être :
  ```
  ## Premier exemple de Makefile
  # il faut generer l'executable toto
  all: toto

  # Le fichier produit.o dépend de produit.c et de produit.h
  produit.o: produit.c produit.h
        gcc -g3 -Wall -c produit.c

  # Le fichier main.o dépend de main.c et de produit.h
  main.o: main.c produit.h
        gcc -g3 -Wall -c main.c

  # Generation de l'exécutable toto qui dépend des fichiers objets
  toto: main.o produit.o
        gcc -g3 -Wall -o toto main.o produit.o
  ```
  Moyennant l'écriture de ce fichier de configuration, il suffira ensuite d'exécuter la commande `make` pour lancer la compilation du programme toto !

8.5.3 Utilisation de macros et de variables

On suppose maintenant qu'on souhaite en plus que les fichiers de header soient placés dans le répertoire `include/` et les fichiers objets dans le répertoire `obj/`. Les fichiers sources (`.c`) sont quand à eux dans le répertoire courant et on souhaite qu'il en soit de même pour l'exécutable final (`toto`).
L'arborescence général des fichiers et le graphe des dépendances entre les fichiers est exposé dans la figure 8.1.
Avec une telle arborescence, on peut modifier le `Makefile` précédent pour gérer la compilation du programme `toto` de la façon suivante :

```
## Deuxieme exemple de Makefile - gestion d'une arborescence
all: toto

obj/produit.o: produit.c include/produit.h
        gcc -g3 -Wall -Iinclude/ -c produit.c -o obj/produit.o

obj/main.o: main.c include/produit.h
        gcc -g3 -Wall -Iinclude/ -c main.c -o obj/main.o

toto: obj/main.o obj/produit.o
```

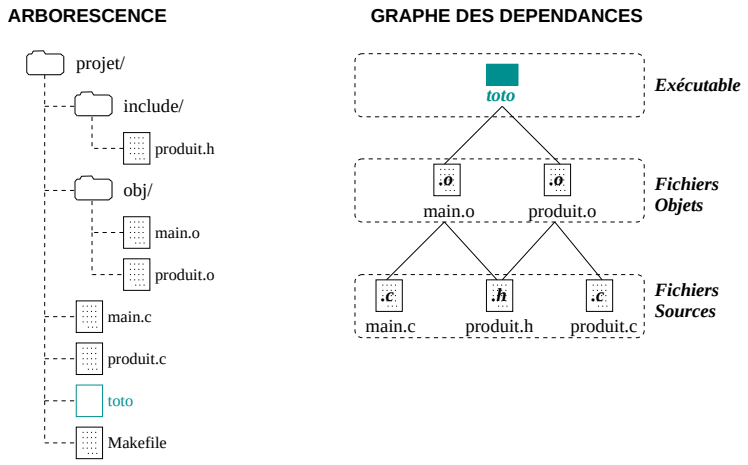


FIG. 8.1 – Arborescence et graphe de dépendances entre les fichiers

```
gcc -g3 -Wall -o toto obj/main.o obj/produit.o
```

On voit qu'un certain nombre d'informations sont redondantes. En particulier, si on change le nom d'un répertoire, il faut le changer à plusieurs endroit dans le `Makefile` d'ou l'idée de centraliser ce type d'informations. On utilise pour cela des variables :

```
## Troisieme exemple de Makefile - gestion d'une arborescence
#  Utilisation de variables
EXE = toto
OBJ_DIR = obj
H_DIR = include
OPT = -g3 -Wall

all: $(EXE)

$(OBJ_DIR)/produit.o: produit.c $(H_DIR)/produit.h
        gcc $(OPT) -Iinclude/ -c produit.c -o $(OBJ_DIR)/produit.o

$(OBJ_DIR)/main.o: main.c $(H_DIR)/produit.h
        gcc $(OPT) -Iinclude/ -c main.c -o $(OBJ_DIR)/main.o

$(EXE): $(OBJ_DIR)/main.o $(OBJ_DIR)/produit.o
        gcc $(OPT) -o $(EXE) $(OBJ_DIR)/main.o $(OBJ_DIR)/produit.o
```

Mais la encore, on continue à dupliquer un certain nombre d'informations. Pour cela, il existe un certain nombre de variables prédéfinies, dont les les plus utiles sont résumées dans le tableau 8.1.
A partir de ces variables, on obtient une version beaucoup plus simple du Makefile précédent :

```
## Quatrieme exemple de Makefile - gestion d'une arborescence
#  Utilisation des variables predefinies
EXE = toto
OBJ_DIR = obj
H_DIR = include
```

`$@`	Le fichier cible (par exemple : `obj/main.o`)
`$*`	Le fichier cible sans suffixe (ex : `obj/main`)
`$<`	Le premier fichier de la liste des dépendances (par exemple : `main.c`)
`$?`	L'ensemble des fichiers de dépendances

TAB. 8.1 – Les principales variables prédéfinies dans un `Makefile`

```
OPT = -Wall -g3

all: $(EXE)

$(OBJ_DIR)/produit.o: produit.c $(H_DIR)/produit.h
        gcc $(OPT) -I$(H_DIR)/ -c $< -o $@

$(OBJ_DIR)/main.o: main.c $(H_DIR)/produit.h
        gcc $(OPT) -I$(H_DIR)/ -c $< -o $@

$(EXE): $(OBJ_DIR)/main.o $(OBJ_DIR)/produit.o
        gcc $(OPT) -o $(EXE) $?
```

A noter que pour déterminer facilement les dépendances entre les différents fichiers, on peut utiliser l'option `-MM` de `gcc`. Par exemple,

```
% gcc -MM -Iinclude/ *.c
main.o: main.c include/produit.h
produit.o: produit.c include/produit.h
```

On rajoute habituellement dans un fichier `Makefile` une cible appelée `clean` permettant de détruire tous les fichiers objets et une cible `distclean` qui supprime également les exécutables générés lors de la compilation.

```
clean:
        rm -f $(OBJ_DIR)/*.o
distclean: clean
        rm -f $(EXE)
```

La commande `make clean` permet donc de "nettoyer" le répertoire courant. Notons que l'on utilise ici la commande `rm` avec l'option `-f` qui évite l'apparition d'un message d'erreur si le fichier à détruire n'existe pas.

Il y aurait encore beaucoup de choses à dire sur les `Makefile` mais cela dépasse largement le cadre de ce polycopié. La meilleur documentation à ce sujet se trouve à l'URL `http://www.gnu.org/software/make/manual/make.html`.
Pour conclure, on notera que tous les exemples de Makefile proposés doivent être modifiés dès lors qu'un nouveau fichier est ajouté au projet. Un exemple de Makefile générique (dans le sens où il n'a pas besoin d'être modifié lors de l'ajout de nouveau fichiers) est fourni an annexe B page 112. Pour réutiliser ce Makefile dans n'importe quel projet, il suffit d'adapter les variables `$(INCL)`, `$(DIR_OBJ)` et `$(EXE)`. Pour plus d'informations, il suffit de taper `make help`.

Chapitre 9

La bibliothèque standard

Ce chapitre est un aide-mémoire qui donne la liste exhaustive de toutes les fonctions de la bibliothèque standard, accompagnées la plupart du temps de leurs prototypes.
Les fichiers d'en-tête de la librairie ANSI sont les suivants (ceux correspondants à de nouvelles fonctionnalités introduites dans la norme ANSI C99 sont indiquées par un astérisque entre parenthèses (*).

`assert.h`	`inttypes.h`(*)	`signal.h`	`stdlib.h`
`complex.h`(*)	`iso646.h`(*)	`stdarg.h`	`string.h`
`ctype.h`	`limits.h`	`stdbool.h`(*)	`tgmath.h`(*)
`errno.h`	`locale.h`	`stddef.h`	`time.h`
`fenv.h`(*)	`math.h`	`stdint.h`(*)	`wchar.h`(*)
`float.h`	`setjmp.h`	`stdio.h`	`wctype.h`(*)

De manière générale, pour obtenir le maximum d'informations sur une commande, on utilisera la commande `man`.

9.1 Diagnostics d'erreurs `<assert.h>`

Cette bibliothèque ne fournit qu'une seule fontion, `assert`, qui permet de mettre des assertions dans le source du programme. Son prototype est le suivant :
`void assert(int expression);`
À l'exécution, si le paramètre de `assert` s'évalue à faux, le programme est stoppé sur terminaison anormale. Exemple :

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

#define TAILLE 512

int calcul(int tab[], int i) {
    assert(i >= 0 && i < TAILLE);
    return 2 * tab[i] + 5;
}
```

```
int main() {
  int tableau[TAILLE];
#ifndef NDEBUG
  printf("DEBUG: tableau[0] = %d\n", tableau[0]);
#endif
  printf("%d\n", calcul(tableau, 1024));
  return 0;
}
```

L'exécution de ce programme renverra :

```
DEBUG: tableau[0] = 180613
a.out: toto.c:7: calcul: Assertion ‘i >= 0 && i < 512’ failed.
Abandon
```

9.2 Gestion des nombres complexes <complex.h>

Cette bibliothèque permet de gérer les nombres complexes, de la forme $z = a + i * b$, où $a, b \in \mathbb{R}$. On rappelle que a est la partie réelle de z, b sa partie imaginaire et i le nombre imaginaire $i = \sqrt{-1}$ ($i^2 = -1$).
`man complex` pour des exemples de code utilisant ces nombres complexes.

9.3 Classification de caractères et changements de casse <ctype.h>

Toutes ces fonctions permettent de tester une propriété sur le caractère passé en paramètre :

Fonction	Prototype	Teste si le paramètre est
isalnum	int isalnum(int c)	une lettre ou un chiffre
isalpha	int isalpha(int c)	une lettre
isblank	int isblank(int c)	un espace ou une tabulation
iscntrl	int iscntrl(int c)	un caractère de commande
isdigit	int isdigit(int c)	un chiffre décimal
isgraph	int isgraph(int c)	un caractère imprimable ou le blanc
islower	int islower(int c)	une lettre minuscule
isprint	int isprint(int c)	un caractère imprimable (pas le blanc)
ispunct	int ispunct(int c)	un signe de ponctuation
isspace	int isspace(int c)	un caractère d'espace blanc
isupper	int isupper(int c)	une lettre majuscule
isxdigit	int isxdigit(int c)	un chiffre hexadécimal

On dispose également de deux fonctions de conversions majuscules/minuscules :

Fonction	Prototype	Action
tolower	int tolower(int c)	convertit c en minuscule
toupper	int toupper(int c)	convertit c en majuscule.

9.4 Valeur du dernier signal d'erreur <errno.h>

Cette bibliothèque ne définit qu'une variable globale,
`extern int errno;`
dont la valeur est un numéro d'erreur affecté lors d'appels systèmes (et par certaines fonctions de librairie). Cette valeur n'a de sens que si l'appel système s'est mal passé et retourne une erreur (la valeur -1). Dans ce cas, la variable `errno` indique quel type d'erreur s'est produite.
`man errno` fournit la liste des valeurs d'erreurs valides.

9.5 Gestion d'un environnement à virgule flottante <fenv.h>

Le standard C99 a introduit via `<fenv.h>` la notion d'*environnement à virgule flottante* afin de permettre une représentation plus détaillée des conditions d'erreurs dans l'arithmétique en virgule flottante.
Deux variables systèmes sont définies dans cet environnement :

1. les *flags de statut*, utilisés dans la gestion des exceptions en virgule flottante.
2. les *modes de contrôle* qui déterminent les différents comportements de l'arithmétique en virgule flottante (la méthode d'arrondi utilisée par exemple).

9.5.1 Gestion des exceptions

Toute erreur (exception) en virgule flottante est caractérisée par une constante symbolique :
– `FE_DIVBYZERO` : division par 0 ;
– `FE_INEXACT` : le résultat de l'opération n'est pas exact ;
– `FE_INVALID` : résultat indéfini ;
– `FE_OVERFLOW` : dépassement de capacité ;
– `FE_UNDERFLOW` : sous-dépassement de capacité.
La constante `FE_ALL_EXCEPT` est un OU bit-à-bit de toutes les constantes précédentes.
Les fonctions suivantes sont utilisées pour gérer ces exceptions. Toutes (à l'exception de `fetestexcept`) retournent 0 en cas de succès. L'argument `excepts` indique l'une des constantes précédentes.

Fonction	Prototype	Action
`feclearexcept`	`int feclearexcept(int excepts)`	Efface les exceptions `excepts`
`fegetexceptflag`	`int fegetexceptflag(fexcept_t *flagp, int excepts)`	Sauvegarde le statut d'exception spécifié dans `*flagp`
`feraiseexcept`	`int feraiseexcept(int excepts)`	Déclenche l'exception `expects`
`fesetexceptflag`	`int fesetexceptflag(fexcept_t *flagp, int excepts)`	Sauvegarde le statut de l'exception `expects` dans `*flagp`
`fetestexcept`	`int fetestexcept(int excepts)`	renvoie les bits correspondants aux exceptions courantes

9.5.2 Gestion des arrondis

`fenv.h` supporte quatre méthodes d'arrondis, caractérisées par les constantes symboliques suivantes :
– `FE_DOWNWARD` : arrondi vers la valeur inférieure la plus proche ;
– `FE_TONEAREST` : arrondi vers la valeur la plus proche ;
– `FE_TOWARDZERO` : partie entière ;
– `FE_UPWARD` : arrondi vers la valeur supérieure la plus proche.
Les fonctions `int fegetround()` et `int fesetround( int rounding_mode)` permettent de lire et de changer la méthode d'arrondi courante.

9.5.3 Gestion des environnements en virgule flottante.

L'environnement de travail en virgule flottante peut être manipulé via les fonctions suivantes sous forme d'un seul objet opaque de type `fenv_t`. L'environnement par défaut est représenté par la constante `FE_DFL_ENV`.

Fonction	Prototype	Action
`fegetenv`	`int fegetenv(fenv_t *envp)`	Sauve l'environnement courant dans `*envp`
`feholdexcept`	`int feholdexcept(fenv_t *envp)`	idem, et efface tous les flags d'exceptions et continue en mode "sans interruption"
`fesetenv`	`int fesetenv(fenv_t *envp)`	recharge l'environnement à `*envp`
`feupdateenv`	`int feupdateenv(fenv_t *envp)`	idem et replace les exceptions déjà présentes dans `*envp`

9.6 Intervalle et précision des nombres flottants `<float.h>`

`float.h` définit des constantes symboliques qui caractérisent les types réels, en particulier :

`FLT_DIG`	nombre de chiffres significatifs d'un `float`
`DBL_DIG`	nombre de chiffres significatifs d'un `double`
`LDBL_DIG`	nombre de chiffres significatifs d'un `long double`
`FLT_EPSILON`	le plus petit nombre tel que `1.0 + x != 1.0`
`DBL_EPSILON`	le plus petit nombre tel que `1.0 + x != 1.0`
`LDBL_EPSILON`	le plus petit nombre tel que `1.0 + x != 1.0`
`FLT_MAX`	le plus grand nombre du type `float`
`DBL_MAX`	le plus grand nombre du type `double`
`LDBL_MAX`	le plus grand nombre du type `long double`
`FLT_MIN`	le plus petit nombre du type `float`
`DBL_MIN`	le plus petit nombre du type `double`
`LDBL_MIN`	le plus petit nombre du type `long double`

9.7 Définitions de types entiers de taille fixée `<inttypes.h>`

La bibliothèque `<inttypes.h>` définit au moins les types suivants :

`int8_t`	type entier signé sur 8 bits
`int16_t`	type entier signé sur 16 bits
`int32_t`	type entier signé sur 32 bits
`int64_t`	type entier signé sur 64 bits
`uint8_t`	type entier non signé sur 8 bits
`uint16_t`	type entier non signé sur 16 bits
`uint32_t`	type entier non signé sur 32 bits
`uint64_t`	type entier non signé sur 64 bits
`intptr_t`	entier signé pouvant stocker un pointeur (une adresse)
`uintptr_t`	entier non signé pouvant stocker un pointeur (une adresse)

En pratique, ces types sont définis dans `stdint.h` (voir §9.17).

9.8 Alias d'opérateurs logiques et binaires `<iso646.h>`

`<iso646.h>` définit des alias (synonymes) aux opérateurs binaires et logiques :

`and`	$\Longleftrightarrow$	`&&`	`and_eq`	$\Longleftrightarrow$	`&=`
`bitand`	$\Longleftrightarrow$	`&`	`bitor`	$\Longleftrightarrow$	`|`
`compl`	$\Longleftrightarrow$	`~`	`not`	$\Longleftrightarrow$	`!`
`not_eq`	$\Longleftrightarrow$	`!=`	`or`	$\Longleftrightarrow$	`||`
`or_eq`	$\Longleftrightarrow$	`|=`	`xor`	$\Longleftrightarrow$	`^`
`xor_eq`	$\Longleftrightarrow$	`^=`			

9.9 Intervalle de valeur des types entiers `<limits.h>`

Le fichier `<limits.h>` définit les constantes symboliques qui précisent les valeurs maximales et minimales qui peuvent être représentées par un type entier donné.

Type	Minimum	Maximum	Maximum (types non-signés)
`char`	`CHAR_MIN`	`CHAR_MAX`	`UCHAR_MAX`
`signed char`	`SCHAR_MIN`	`SCHAR_MAX`	
`short`	`SHRT_MIN`	`SHRT_MAX`	`USHRT_MAX`
`int`	`INT_MIN`	`INT_MAX`	`UINT_MAX`
`long`	`LONG_MIN`	`LONG_MAX`	`ULONG_MAX`
`long long`	`LLONG_MIN`	`LLONG_MAX`	`ULLONG_MAX`

`<limits.h>` définit également la constanste `CHAR_BIT`, correspondant au nombre de bits du type `char`.

9.10 Gestion de l'environnement local `<locale.h>`

Il y a deux fonctions permettant de gérer les conventions nationales concernant l'écriture du point décimal dans les nombres, le signe représentant l'unité monétaire etc... Ces fonctions sont `setlocale` et `localeconv` (faire un `man` sur ces fonctions pour plus de détails).

9.11 Les fonctions mathématiques de <math.h>

Pour utiliser les fonctions de cette librairie, il faut en plus de l'inclusion de la librairie par la directive `#include <math.h>` ajouter l'option `-lm` a l'édition de lien. Ainsi, la compilation de `prog.c` utilisant la librairie `<math.h>` se fera par la commande : `gcc -g3 -Wall -lm prog.c -o prog`
Le résultat et les paramètres de toutes ces fonctions sont de type `double`. Si les paramètres effectifs sont de type `float`, ils seront convertis en double par le compilateur.

9.11.1 Fonctions trigonométriques et hyperboliques

Fonction	Sémantique	Fonction	Sémantique
`acos`	arc cosinus	`cos`	cosinus
`asin`	arc sinus	`sin`	sinus
`atan`	arc tangente	`tan`	tangente
`cosh`	cosinus hyperbolique	`sinh`	sinus hyperbolique
`tanh`	tangente hyperbolique	`atan2`	arc tangente

9.11.2 Fonctions exponentielles et logarithmiques

Fonction	Sémantique
`exp`	exponentielle
`frexp`	étant donné x, trouve n et p tels que x = n * 2p
`ldexp`	multiplie un nombre par une puissance entière de 2
`log`	logarithme
`log10`	logarithme décimal
`modf`	calcule partie entière et décimale d'un nombre

9.11.3 Fonctions diverses

Fonction	Sémantique
`ceil`	entier le plus proche par les valeurs supérieures
`fabs`	valeur absolue
`floor`	entier le plus proche par les valeurs inférieures
`fmod`	reste de division
`pow`	puissance
`sqrt`	racine carrée

9.12 Branchements non locaux <setjmp.h>

L'instruction `goto` présentée au §2.2.6 ne permet de réaliser des branchements qu'au sein d'une même procédure. Pour réaliser des branchements à l'extérieur d'une procédure, il faut utiliser `setjmp` et `longjmp`.
Prototype de ces fonctions :

```
int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
```

Plus précisément, les fonctions `setjmp` et `longjmp` permettent de gérer les erreurs et les interruptions rencontrées dans des routines bas-niveau. `setjmp` sau-

vegarde le contexte de pile et d'environnement dans `env` afin de l'utiliser ultérieurement avec `longjmp`.
Pour plus de détails, utiliser le **man** sur ces fonctions.

9.13 Manipulation des signaux <signal.h>

Lorsqu'un évènement exceptionnel se produit, le système d'exploitation peut envoyer un signal aux processus. Ceci peut arriver en cas d'erreur grave (une erreur d'adressage par exemple). Le tableau suivant fournit la liste de quelques constantes symboliques définies dans `<signal.h>` et caractérisant ces erreurs :

Signal	Signification
SIGABRT	Fin anormale du programme (ex : utilisation de la fonction `abort()`)
SIGFPE	Exception à la virgule flottante.
SIGILL	Instruction illégale rencontrée dans le code machine.
SIGINT	Interruption par la touche `Ctrl-C`.
SIGSEGV	Violation de Segmentation : accès illégal à la mémoire.
SIGTERM	Terminer : requête pour terminer le programme.

Deux fonctions permettent d'interagir avec le mécanisme des signaux :

Fonct.	Prototype	Action
raise	`int raise(int sig)`	Envoie le signal `sig` au processus exécutant.

signal	`typedef void (*sigh_t)(int);`
	`sigh_t signal(int signum,`
	`sigh_t handler);`
		précise la réponse du programme au signal `signum`. `handler` est un pointeur vers la fonction effectuant le traitement de ce signal.

Comme toujours, le `man` reste la meilleur source d'informations sur ce sujet.

9.14 Nombre variable de paramètres <stdarg.h>

Si on désire programmer une fonction avec un nombre variable de paramètres (comme `printf`), on dispose des macros `va_start`, `va_arg` et `va_end`.
L'utilisation de ces macros et du type associé `va_list` est abordé en détail dans le §5.7 page 67.

9.15 Définition du type booléen <stdbool.h>

La librairie `<stdbool.h>`, ajoutée avec la norme ANSI C99, introduit la notion de booléen et définit le type booléen `bool` et les valeurs associées, `true` et `false`.

9.16 Définitions standards <stddef.h>

Cette librairie introduit un ensemble de types prédéfinis très utiles qui sont listés dans le tableau suivant :

Type	Description
ptrdiff_t	Différence entre deux pointeurs ($\simeq$ `int` en général)
wchar_t	Employé pour les caractères étendus (voir §9.23)
size_t	Taille d'un objet en nombres d'octets ($\simeq$ `unsigned int` en général)

<stddef.h> définit également le pointeur NULL, utilisé lors de l'initialisation des pointeurs (cf chapitre 3) ainsi que la fonction :
`size_t offsetof(type, member-designator)`
qui permet de connaître le décalage (en nombre d'octets) entre un champ d'une structure (`member-designator`) et le début de cette structure (`type`). Ainsi, si on suppose définie la structure `toto` ayant un champ `next` (voir §4.3.5), le décalage du champ `data` est donné par l'expression :
`offsetof (struct toto, data)`

9.17 Définitions de types entiers <stdint.h>

La librairie <stdint.h> est un sous ensemble de la librairie <inttypes.h> (voir §9.7) et en constitue donc une version allégée, adaptée aux environnements embarqués qui peuvent ne pas supporter les fonctions d'entrées/sorties formattées.

9.18 Entrées-sorties <stdio.h>

9.18.1 Manipulation de fichiers

Fonction	Prototype	Action (voir chapitre 6)
fopen	`FILE *fopen(char *path, char *mode)`	ouverture d'un fichier
fclose	`int fclose(FILE *fp)`	Fermeture d'un fichier
fflush	`int fflush(FILE *stream)`	écriture des buffers en mémoire dans le fichier
tmpfile	`FILE *tmpfile (void)`	création d'un fichier temporaire

9.18.2 Entrées et sorties formatées

Fonction	Prototype	Action (voir §2.5/chapitre 6)
fprintf	`int fprintf(FILE *stream, char *format, ...)`	écriture sur un fichier
fscanf	`int fscanf(FILE *stream, char *format, ...)`	lecture depuis un fichier
printf	`int printf(char *format, ...)`	écriture sur la sortie standard
scanf	`int scanf(char *format, ...)`	lecture depuis l'entrée standard
sprintf	`int sprintf(char *s, char *format, ...)`	écriture dans la chaîne de caractères `s`
sscanf	`int sscanf(char *s, char *format, ...)`	lecture depuis la chaîne de caractères `s`

9.18.3 Impression et lecture de caractères

Fonction	Prototype	Action
fgetc	`int fgetc(FILE *stream)`	lecture d'un caractère depuis un fichier
fputc	`int fputc(int c, FILE *stream)`	écriture d'un caractère sur un fichier
getc	`int getc(FILE *stream)`	équivalent de fgetc mais optimisée
putc	`int putc(int c, FILE *stream)`	équivalent de fputc mais optimisée
getchar	`int getchar(void)`	lecture d'un caractère depuis `stdin`
putchar	`int putchar(int c)`	écriture d'un caractère sur `stdout`
fgets	`char *fgets(char *s, FILE *stream)`	lecture d'une chaîne de caractères depuis un fichier
fputs	`int *fputs(char *s, FILE *stream)`	écriture d'une chaîne de caractères sur un fichier
gets	`char *gets(char *s)`	lecture d'une chaîne de caractères sur `stdin`
puts	`int *puts(char *s)`	écriture d'une chaîne de caractères sur `stdout`

9.19 Utilitaires divers <stdlib.h>

9.19.1 Allocation dynamique

Ces fonctions sont décrites dans le chapitre 3 et au §3.5.

Fonct.	Prototype	Action
calloc	void *calloc(size_t n, size_t size)	allocation dynamique de n*size octects et initialisation à zéro.
malloc	void *malloc(size_t size)	allocation dynamique
realloc	void *realloc(void *ptr, size_t size)	modifie la taille d'une zone préalablement allouée par calloc ou malloc
free	void free(void *ptr)	libère une zone mémoire

9.19.2 Conversion de chaînes de caractères en nombres

Les fonctions suivantes permettent de convertir une chaîne de caractères en un nombre.

Fonct.	Prototype	Action
atof	double atof(char *chaine)	convertit chaine en un double
atoi	int atoi(char *chaine)	convertit chaine en un int
atol	long atol(char *chaine)	convertit chaine en un long int

Il semblerait que qu'il faille préférer maintenant les fonctions strtol, strtoll et strtoq.

9.19.3 Génération de nombres pseudo-aléatoires

La fonction rand fournit un nombre entier pseudo-aléatoire dans l'intervalle [0,RAND_MAX]. RAND_MAX est une constante prédéfinie au moins égale à $2^{15} - 1$. L'aléa fournit par la fonction rand n'est toutefois pas de très bonne qualité.
La valeur retournée par rand dépend de l'initialisation (germe ou seed en anglais) du générateur. Cette dernière est égale à 1 par défaut mais elle peut être modifiée à l'aide de la fonction srand.

Fonct.	Prototype	Action
rand	int rand(void)	fournit un nombre pseudo-aléatoire
srand	void srand(unsigned int seed)	modifie la valeur du germe du générateur pseudo-aléatoire

On utilise souvent la valeur de l'horloge interne de l'ordinateur (obtenue à l'aide de la librairie <time.h>, voir §9.22) pour fournir un "bon"[1] germe à srand. Exemple :

```
#include <stdlib.h>   //see also 'man 3 rand'
#include <time.h>

/* initialize random generator */
void init_rand(){ srand(time(NULL)); }
```

[1] Pas suffisamment bon cependant pour les applications cryptographiques

```
/* return a random number between 1 and m */
unsigned long myRand(unsigned long m){
    return 1+(unsigned long)(((double) m)*rand()/(RAND_MAX+1.0));
}
```

9.19.4 Arithmétique sur les entiers

Fonct.	Prototype	Action
`abs`	`int abs(int n)`	valeur absolue d'un entier
`labs`	`long labs(long n)`	idem mais pour un long int
`div`	`div_t div(int a, int b)`	quotient et reste de la division euclidienne de `a` par `b`.
`ldiv`	`ldiv_t ldiv(long a, long b)`	idem pour les `long int`

Les structures `div_t` et `ldiv_t` disposent de deux champs, `quot` et `rem`, qui stockent les résultats des fonctions `div` et `ldiv`.

9.19.5 Recherche et tri

Les fonctions `qsort` (resp. `bsearch`) permettent de trier un tableau (resp. rechercher un élément dans un tableau déjà trié). Pour leurs syntaxes : voir §5.8.

9.19.6 Communication avec l'environnement

Fonct.	Prototype	Action
`abort`	`void abort(void)`	terminaison anormale du programme
`exit`	`void exit(int etat)`	terminaison du programme ; rend le contrôle au système en lui fournissant la valeur etat (0 = fin normal).
`system`	`int system(char *s)`	exécution de la commande système définie par `s`.

9.20 Manipulation de chaînes de caractères <string.h>

Fonct.	Prototype	Action
strcpy	char *strcpy(char *ch1,char *ch2)	copie ch2 dans ch1 ; retourne ch1.
strncpy	char *strcpy(char *ch1,char *ch2, int n)	idem mais ne copie que **n** caractères.
strcat	char *strcat(char *ch1,char *ch2)	copie ch2 à la fin de ch1 ; retourne ch1.
strncat	char *strncat(char *ch1,char *ch2, int n)	idem mais seuls **n** caractères sont copiés
strcmp	int strcmp(char *ch1,char *ch2)	compare ch1 et ch2 pour l'ordre lexicographique ;
strncmp	int strcmp(char *ch1,char *ch2, int n)	idem mais seuls **n** caractères sont comparés.
strchr	char *strchr(char *ch,char c)	retourne un pointeur sur la première occurence de c dans ch et NULL si c $\notin$ ch.
strrchr	char *strchr(char *chaine,char c)	retourne un pointeur sur la dernière occurence de c dans ch et NULL si c $\notin$ ch
strstr	char *strchr(char *ch1,char *ch2)	retourne un pointeur sur la première occurence de ch2 dans ch1 et NULL si ch2 $\notin$ ch1
strlen	int strlen(char *chaine)	retourne la longueur de chaine.

La librairie <string.h> fournit un ensemble de fonctions permettent de gérer les chaînes de caractères pour :

- copier : memcpy, memmove, strcpy, strncpy ;
- concaténer : strcat, strncat ;
- comparer : memcmp, strcmp, strcoll, strncmp ;
- transformer : strxfrm ;
- rechercher : memchr, strchr, strcspn, strpbrk, strrchr, strspn, strstr, strtok ;
- initialiser : memset ;
- mesurer : strlen ;
- obtenir un message d'erreur à partir du numéro de l'erreur : strerror.

Concernant strcmp, on notera que le résultat de cette fonction est

- une valeur négative si ch1 est inférieure à ch2 (pour l'ordre lexicographique) ;
- 0 si ch1 et ch2 sont identiques ;
- une valeur positive sinon.

9.21 Macros génériques pour les fonctions mathématiques <tgmath.h>

Cette librairie, introduite avec la norme ANSI C99, est un sur-ensemble des librairies <math.h> et de <complex.h> et définit pour chacune des fonctions communes à ces deux librairie une version similaire définie sous forme de macro pour un type générique. La liste de ces fonctions est donnée dans le tableau suivant :

<math.h>	<complex.h>	Type-générique
Fonction	**Function**	**Macro**
acos()	cacos()	acos()
asin()	casin()	asin()
atan()	catan()	atan()
acosh()	cacosh()	acosh()
asinh()	casinh()	asinh()
atanh()	catanh()	atanh()
cos()	ccos()	cos()
sin()	csin()	sin()
tan()	ctan()	tan()
cosh()	ccosh()	cosh()
sinh()	csinh()	sinh()
tanh()	ctanh()	tanh()
exp()	cexp()	exp()
log()	clog()	log()
pow()	cpow()	pow()
sqrt()	csqrt()	sqrt()
fabs()	cabs()	fabs()

Pour un complément d'informations :
`http://www.opengroup.org/onlinepubs/009695399/basedefs/tgmath.h.html`

9.22 Date et heure <time.h>

Plusieurs fonctions permettent d'obtenir la date et l'heure. Le temps est représenté par des objets de type `time_t` ou `clock_t`, lesquels correspondent généralement à des `int` ou à des `long int`. Les fonctions de manipulation de la date et de l'heure sont : `clock`, `difftime`, `mktime`, `time`, `asctime`, `ctime`, `gmtime`, `localtime`, `strftime`.
Les plus utiles sont listées dans le tableau suivant :

Fonct.	Prototype	Action
`time`	`time_t time(time_t *tp)`	retourne dans `*tp` le nombre de secondes écoulées depuis le 1er janvier 1970, 0 heures G.M.T.
`difftime`	`double difftime(time_t t1, time_t t2)`	retourne la différence `t1 - t2` en secondes.
`ctime`	`char *ctime(time_t *tp)`	convertit `*tp` en une chaîne de caractères explicitant la date et l'heure (format prédéterminé).
`clock`	`clock_t clock(void)`	renvoit le temps CPU (ms) depuis le dernier `clock`.

9.23 Manipulation de caractères étendus <wchar.h> et <wctype.h>

Le type `char` est codé sur 8 bits et le restera parce qu'il désigne la plus petite unité de données adressable.
Dans le cadre du design d'applications multilingues, on devra donc gérer des caractères et des chaînes au format UNICODE.

A cet effet, la norme ANSI C99 a introduit :
– un type de caractère codé sur 16 bits `wchar_t` ;
– un ensemble de fonctions similaires à celles contenues dans `<string.h>` et `<ctype.h>` (déclarées respectivement dans `<wchar.h>` et `<wctype.h>`) ;
– un ensemble de fonctions de conversion entre les types `char *` et `wchar_t *` (déclarées dans `<stdlib.h>`).

Pour plus de détails :
`http://www.opengroup.org/onlinepubs/007908799/xsh/wchar.h.html` ou
`http://www.freenix.fr/unix/linux/HOWTO/Unicode-HOWTO-5.html`

Annexe A

Etude de quelques exemples

A.1 Gestion des arguments de la ligne de commande

Le code source suivant traite l'exemple présenté au §4.2.6 page 50.
Commande de compilation : `gcc -O3 -Wall commande.c -o commande`

```
/**
 * @file   command.c
 * @author Sebastien Varrette <Sebastien.Varrette@imag.fr>
 * @date   Tue Oct  4 2005
 *
 * @brief  see command.h
    traitement des arguments de la ligne
    de commande utilisant la librairie <getopt.h>
    Exemple traité: programme 'command'  au format d'appel
    suivant  (les éléments entre crochets sont optionnels):

   ./command [-i arg_i] [-f arg_f] [-s arg_s] [-k arg_k] [-h] [-V] arg_file

   Les options de la ligne de commande sont donc:
     -i: permet de spécifier la valeur de arg_i, de type int
         Valeur par défaut: 14
     -f: permet de spécifier la valeur de arg_f, de type float
         Valeur par défaut: 1.5
     -s: permet de spécifier la valeur de arg_s, une chaîne de caractères
         possédant au plus MAX_SIZE caractères
         Valeur par défaut: "On est super content!"
     -h: affiche l'aide (auteur, but du programme et usage) et sort du
         programme.
     -V: affiche la version du programme et sort

   arg_file est un paramètre obligatoire du programme qui spécifie
       la valeur de la chaîne de caractères arg_file.
 */
 /*****************************************************************************/
#include <stdio.h>     /* for printf */
#include <stdlib.h>    /* for exit  */
#include <assert.h>    /* for assert */
#include <string.h>
#include <getopt.h>

//#include "toto.h"
```

```
#define VERSION  0.1   // source version
#define MAX_SIZE 256   // maximal size of strings, including '\0'

void printHelp(char * command);                         // print help message
void printError(char * error_message, char * command); // print error message
void printVersion(char * command);                      // print version

/**
 * Entry point where the program begins its execution.
 * @param argc number of arguments of the command line
 * @param argv multi-array containing the command line arguments
 *             (argv[0] is the name of the program, argv[argc]=NULL)
 * @return status of the execution (0: correct execution)
 */
int main(int argc, char * argv[]){

    // Command line management
    extern char *optarg;  // specific to getopt
    extern int optind;    //, opterr, optopt;  // id.
    char c;

    // to deal with options
    char opt_i[MAX_SIZE] = "";
    char opt_f[MAX_SIZE] = "";
    long arg_i = 14;
    double arg_f = 1.5;
    char arg_s[MAX_SIZE] = "On est super content!";
    char arg_file[MAX_SIZE] = "";


    // Management of parameters in command line
    while ((c = getopt(argc, argv, "i:f:s:hV")) != -1) {
        switch(c) {
        case 'h': printHelp(argv[0]);    return EXIT_SUCCESS;
        case 'V': printVersion(argv[0]); return EXIT_SUCCESS;
        case 'i':
            assert(strlen(optarg) < MAX_SIZE); // check size
            strncpy(opt_i,optarg,MAX_SIZE);    // copy current arg to opt_i
            // Minimal check for validity (opt_i isn't a command line option)
            if (opt_i[0] == '-') printError("Bad Format for arg_i!",argv[0]);
            arg_i = strtol(opt_i, NULL, 10);   // last arg: base for convertion
            break;
        case 'f':
            assert(strlen(optarg) < MAX_SIZE); // check size
            strncpy(opt_f,optarg,MAX_SIZE);    // copy current arg to opt_f
            // Minimal check for validity (opt_f isn't a command line option)
            if (opt_f[0] == '-') printError("Bad Format for arg_f!",argv[0]);
            arg_f = strtod(opt_f, NULL);       // conversion
            break;
        case 's':
            assert(strlen(optarg) < MAX_SIZE); // check size
            strncpy(arg_s,optarg,MAX_SIZE);    // copy current arg to arg_s
            if (arg_s[0] == '-') printError("Bad Format for arg_i!",argv[0]);
            break;
        default: printError("Bad Format!",argv[0]);
        }
```

```
    }

    // Now proceed to detect errors and/or set the values

    // parameter arg_file is required. If not, print error and exit
    // argc - optind == 0 : no arg_file
    // argc - optind > 1 : to many parameters or error not detected
    if ((argc - optind) != 1) printError("Bad Format",argv[0]);

    // required parameter: arg_file
    assert(strlen(argv[optind]) < MAX_SIZE);
    strncpy(arg_file,argv[optind],MAX_SIZE);

    //Print values
    printf("arg_i = %ld\n",arg_i);
    printf("arg_f = %f\n",arg_f);
    printf("Valeur de arg_s: %s\n",arg_s);
    printf("Valeur de arg_file: %s\n",arg_file);

    return EXIT_SUCCESS;
}


/**
 * Print the help message
 * @param command used (argv[0])
 */
void printHelp(char * command) {
    printf("NAME\n"                                                          );
    printf("      %s\n",command                                              );
    printf("\nSYNOPSIS\n"                                                    );
    printf("      %s [-h] [-V]\n",command                                    );
    printf("      %s [-i arg_i] [-f arg_f] [-s arg_s] arg_file\n",command     );
    printf("\nDESCRIPTION\n"                                                 );
    printf("      -h : print help and exit\n"                                );
    printf("      -V : print version and exit\n"                             );
    printf("      -i arg_i : set arg_i (long)\n"                             );
    printf("           Default value : 14\n"                                 );
    printf("      -f arg_f : set arg_f (float)\n"                            );
    printf("           Default value : 1.5\n"                                );
    printf("      -s arg_s : set arg_s (char *), a string with at most MAX_SIZE\n");
    printf("           characters. Default value : \"Toto\"\n"               );
    printf("\nAUTHOR\n"                                                      );
    printf("      Sebastien Varrette <Sebastien.Varrette@imag.fr>\n"         );
    printf("      Web page : http://www-id.imag.fr/~svarrett/\n"             );
    printf("\nREPORTING BUGS\n"                                              );
    printf("      Please report bugs to <Sebastien.Varrette@imag.fr>\n"      );
    printf("\nCOPYRIGHT\n"                                                   );
    printf("      This  is free software; see the source for copying conditions.\n");
    printf("      There is NO warranty; not even for MERCHANTABILITY or FITNESS\n");
    printf("      FOR A PARTICULAR PURPOSE."                                 );
    printf("\nSEE ALSO\n"                                                    );
    printf("      http://www-id.imag.fr/~svarrett/perso.html\n"              );
}

/**
 * print error message on stderr
```

```
 * @param error_message Error message to print
 * @param command       command used (argv[0])
 */
void printError(char * error_message, char * command) {
    fprintf(stderr, "[ERROR] %s\n",error_message);
    fprintf(stderr, "Use '%s -h' for help\n", command);
    exit(EXIT_FAILURE);
}
/**
 * print version
 * @param command used (argv[0])
 */
void printVersion(char * command) {
    fprintf(stderr, "This is %s  version %f\n",command,VERSION);
    fprintf(stderr, "Please type '%s -h' for help\n", command);
}
```

A.2 Exemple de liste chaînée

Le code source suivant traite l'exemple présenté au §4.3.5 page 53.
Commande de compilation : `gcc -O3 -Wall toto.c -o toto`

```
/*******************************************************
 Fichier: toto.c
 Auteur: Sebastien Varrette <Sebastien.Varrette@imag.fr>

 Un exemple de gestion des listes chainées
 *******************************************************/
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>

/* Définition de la structure toto*/
struct toto {
    int data;
    struct toto *next;
};
//type liste synonyme du type pointeur vers une struct toto
typedef struct toto *liste;
/*************************************
 * Insère un élément en tête de liste *
 *************************************/
liste insere(int element, liste Q) {
    liste L;
    L = (liste)malloc(sizeof(struct toto)); // allocation de la zone
                                            // mémoire necéssaire
    L->data = element;
    L->next = Q;
    return L;
}
/*************************************
 * Insère un élément en queue de liste *
 *************************************/
liste insereInTail(int element, liste Q) {
    liste L, tmp=Q;
    L = (liste)malloc(sizeof(struct toto)); // allocation de la zone
                                            // mémoire necéssaire
    L->data = element;
    L->next = NULL;
    if (Q == NULL)
        return L;
    // maintenant tmp=Q est forcement non vide => tmp->next existe
    while (tmp->next != NULL) tmp = tmp->next; // deplacement jusqu'au
                                               // dernier elt de la liste
    tmp->next = L;
    return Q;
}
/*************************************
 * Supprime l'élément en tête de liste *
 *************************************/
liste supprime_tete(liste L) {
    liste suivant = L;
    if (L != NULL) { // pour etre sûr que L->next existe
       suivant= L->next;
```

```
       free(L); //libération de l'espace alloué pour une cellule
    }
    return suivant;
}
/*****************************
 * Supprime toute la liste L *
 *****************************/
liste supprime(liste L) {
    while (L != NULL)
        L = supprime_tete(L); //suppression de la tête de liste
    return L;
}
/***********************************
 * Affiche le contenu de la liste L *
 ***********************************/
void print_liste(liste L) {
    liste tmp = L;
    while (tmp != NULL) { //on aurait pu écrire 'while (tmp)'
        printf("%d \t",tmp->data);
        tmp = tmp->next;
    }
    printf("NULL\n");
}

/*******************************************************/
int main() {
    liste L;
    L = insere(14,insere(2,insere(3,insere(10,NULL))));
    L = insereInTail(15,(insereInTail(14,L)));
    printf("Impression de la liste:\nL = \t");
    print_liste(L);
    printf("Suppression de l'element en tete:\nL = \t");
    L = supprime_tete(L);
    print_liste(L);
    printf("Suppression de la liste:\nL = \t");
    L = supprime(L);
    print_liste(L);
    return 0;
}
```

Annexe B

Makefile générique

```
##############################################################################
# Makefile (configuration file for GNU make - see http://www.gnu.org/software/make/)
#
# Compilation of files written in C/C++ (version 0.5)
#
# Author : Sebastien Varrette <Sebastien.Varrette@imag.fr>
#          Web page : http://www-id.imag.fr/~svarrett/
#
# ----------------------------------------------------------------------------
# This is a generic makefile in the sense that it doesn't require to be
# modified when adding/removing new source files.
# ----------------------------------------------------------------------------
#
# Documentation on the Makefile utility may be found at
#                  http://www.gnu.org/software/make/manual/make.html
#
# Available Commands
# ------------------
# make           : Compile files, binary is generated in the current directory
# make force     : Force the complete re-compilation, even if not needed
# make clean     : Remove backup files generated by emacs and vi
# make realclean : Remove all generated files
# make doc       : Generate Doxygen documentation (in Doc/) see www.doxygen.org
# make help      : print help message
#
############################### Variables Declarations ###############################
# Name of the executable to generate --- TO BE ADAPTED ---
EXE          = toto
# Directory where header files (.h) and object files (.o) will be placed
INCLUDE_DIR  = Include
OBJ_DIR      = Obj
# File entensions for C, C++ and header files
C_EXT        = c
CPP_EXT      = cpp
H_EXT        = h
# Source files
SRC          = $(wildcard *.$(C_EXT) *.$(CPP_EXT))
SRC_H        = $(wildcard *.$(H_EXT) $(INCLUDE_DIR)/*.$(H_EXT))
ALL_SRC      = $(SRC) $(SRC_H)
# Check avalibility of source files
ifeq ($(SRC),)
all:
        @echo "No source files available - I can't handle the compilation"
        @echo "Please check the presence and/or extension of source files "
        @echo "(This makefile is configured to manage *.$(C_EXT) or *.$(CPP_EXT) - "\
                "you may modify variables C_EXT and CPP_EXT to reflect your " \
                "own programming style)"
else
```

```
# Object files
OBJ          = $(patsubst %.$(C_EXT),%.o,$(SRC:.$(CPP_EXT)=.o))
ABSOBJ       = $(OBJ:%.o=$(OBJ_DIR)/%.o)
# Backup files generated by text editors
BACKUP_FILE  = $(shell find . -name "*~")
# Doxygen stuff
DOC_DIR      = Doc
DOXYGEN      = $(shell which doxygen)
DOXYGEN_CONF = .doxygen.conf
YES_ATTRIBUTES := JAVADOC_AUTOBRIEF EXTRACT_ALL EXTRACT_PRIVATE EXTRACT_STATIC \
                  SOURCE_BROWSER GENERATE_MAN
# Compilator configuration
# Detect if you have a C or a C++ project through file extension
ifneq ($(filter %.c,$(SRC)),)
        CXX    = gcc
        YES_ATTRIBUTES := $(YES_ATTRIBUTES) OPTIMIZE_OUTPUT_FOR_C
else
        CXX    = g++
        SPECIAL_CPP_OPTION = -Wno-deprecated
endif
CXXFLAGS     = -g3 -O3 -Wall $(SPECIAL_CPP_OPTION) -I$(INCLUDE_DIR) -c
ADD_OPT      = #-lntl -lgmp -lm  # Optionnal option for the linker
# Specifies the list of directories that make should search
VPATH        = $(INCLUDE_DIR):$(OBJ_DIR)
# dependance file used for make rules
MAKEDEP_FILE = .Makefile.dep
################################ Now starting rules ################################
# Required rule : what's to be done each time
all :   $(MAKEDEP_FILE) $(EXE) TAGS

# Generate TAGS for emacs
TAGS : $(ALL_SRC)
        etags $<
        cp TAGS $(INCLUDE_DIR)/

# Clean Options
clean :
        @echo "Remove backup files generated by emacs and vi"
        rm -f $(BACKUP_FILE)

# Clean everything (including object files, binaries and documentation)
realclean : clean
        @echo "Remove object files"
        rm -f $(ABSOBJ)
        @echo "Remove generated executable"
        rm -f $(EXE)
        @if [ ! -z "$(DOC_DIR)" -a ! -z "$(DOXYGEN)" ]; then \
                echo "Remove documentation ('make doc' to regenerate it)"; \
                rm -rf $(DOC_DIR)/*; \
        fi

# Force re-compilation, even if not required
force :
        touch $(ALL_SRC) $(ABSOBJ)
        @$(MAKE)

# Generate the dependance file
$(MAKEDEP_FILE) : $(ALL_SRC)
        $(CXX) $(SPECIAL_CPP_OPTION) -MM -I$(INCLUDE_DIR) $(SRC) > $@

include $(MAKEDEP_FILE)

# Generic description for compilation of object files
%.o : %.$(C_EXT)
        $(CXX) $(CXXFLAGS) $< -o $(OBJ_DIR)/$@
%.o : %.$(CPP_EXT)
```

```
        $(CXX) $(CXXFLAGS) $< -o $(OBJ_DIR)/$@

# Generation of the final binary (see $(EXE))
$(EXE) : $(OBJ) $(ALL_SRC)
        $(CXX) -g -o $@ $(ABSOBJ) $(ADD_OPT)
        @$(MAKE) help

# Help rule - print help message
help :
        @echo '+--------------------------------------------------------------------------------+'
        @echo '|                                Available Commands                              |'
        @echo '+----------------+---------------------------------------------------------------+'
        @echo '| make           | Compile files, binary is generated in the current directory|'
        @echo '| make force     | Force the complete re-compilation, even if not required    |'
        @echo '| make clean     | Remove cache backup files generated by emacs and vi        |'
        @echo '| make realclean | Remove all generated files (including .o and binary)       |'
        @echo '| make doc       | Generate documentation using doxygen (see www.doxygen.org) |'
        @echo '| make help      | Print help message                                         |'
        @echo '+----------------+---------------------------------------------------------------+'

# Test values of variables - for debug purpose
test :
        @echo "INCLUDE_DIR  = $(INCLUDE_DIR)"
        @echo "OBJ_DIR      = $(OBJ_DIR)"
        @echo "EXE          = $(EXE)"
        @echo "SRC          = $(SRC)"
        @echo "SRC_H        = $(SRC_H)"
        @echo "ALL_SRC      = $(ALL_SRC)"
        @echo "OBJ          = $(OBJ)"
        @echo "BACKUP_FILE  = $(BACKUP_FILE)"
        @echo "CXX          = $(CXX)"
        @echo "CXXFLAGS     = $(CXXFLAGS)"
        @echo "ADD_OPT      = $(ADD_OPT)"
        @echo "DOC_DIR      = $(DOC_DIR)"
        @echo "DOXYGEN      = $(DOXYGEN)"
        @echo "DOXYGEN_CONF = $(DOXYGEN_CONF)"
        @echo "YES_ATTRIBUTES = $(YES_ATTRIBUTES)"
        @echo "MAKEDEP_FILE = $(MAKEDEP_FILE)"

# Documentation generation through doxygen
# First check if the $(DOXYGEN) and the $(DOC_DIR) directory exist
# Then Check $(DOXYGEN_CONF) availability;otherwise,generate one with 'doxygen -s -g'
#    The following attributes should be modified in the generated file:
#    - OUTPUT_DIRECTORY should be set to '$(DOC_DIR)', INPUT to '. $(INCLUDE_DIR)'
#    - $(YES_ATTRIBUTES) attributes should be set to YES
#    - OPTIMIZE_OUTPUT_FOR_C should be set to YES if the project in in C
# Finally, launch documentation generation
doc :
ifeq ($(DOXYGEN),)
        @echo "Please install Doxygen to use this option!"
        @echo "('apt-get install doxygen' under Debian)"
else
        @if [ ! -d ./$(DOC_DIR) ]; then \
                echo "$(DOC_DIR)/ does not exist => creating $(DOC_DIR)/"; \
                mkdir -p ./$(DOC_DIR)/; \
        fi
        @if [ ! -f $(DOXYGEN_CONF) ]; then                                                  \
                echo "I don't found the configuration file for Doxygen ($(DOXYGEN_CONF))";   \
                echo "Now generating one using '$(DOXYGEN) -s -g $(DOXYGEN_CONF)'";          \
                $(DOXYGEN) -s -g $(DOXYGEN_CONF);                                          \
                echo "Now updating OUTPUT_DIRECTORY attribute to '$(DOC_DIR)'";            \
                cat $(DOXYGEN_CONF) | sed -e "s/^\(OUTPUT_DIRECTORY \+= \+\).*/\1$(DOC_DIR)/"\
                  > $(DOXYGEN_CONF);                                                       \
                echo "Now updating INPUT attribute to '. $(INCLUDE_DIR)'";                 \
                cat $(DOXYGEN_CONF) | sed -e "s/^\(INPUT \+= \+\).*/\1. $(INCLUDE_DIR)/"     \
                   > $(DOXYGEN_CONF);                                                      \
```

```
                for attr in $(YES_ATTRIBUTES); do                                       \
                        echo "now updating $$attr to YES";                              \
                        cat $(DOXYGEN_CONF) | sed -e "s/^\($$attr \+= \+\).*/\1YES/"      \
                           > $(DOXYGEN_CONF);                                           \
                done; \
        fi
        $(DOXYGEN) $(DOXYGEN_CONF)
        @echo
        @echo Documentation generated in $(DOC_DIR)/
        @echo May be you can try to execute 'mozilla ./$(DOC_DIR)/html/index.html'
endif
endif
```

Annexe C

Le bêtisier

Cette annexe est une collection des bêtises qu'il faut faire au moins une fois dans sa vie pour être vacciné. L'idée est reprise de [Cas98]
La caractéristique de beaucoup de ces erreurs est de ne pas provoquer de message d'erreur du compilateur, rendant ainsi leur détection difficile. La différence entre le texte correct et le texte erroné est souvent seulement d'un seul caractère. La découverte de telles erreurs ne peut donc se faire que par un examen très attentif du source du programe.

C.1 Erreur avec les opérateurs

C.1.1 Erreur sur une comparaison

Ce que voulait le programmeur :	Comparer `a` et `b`
Ce qu'il aurait dû écrire :	`if (a == b)`
Ce qu'il a écrit :	`if (a = b)`
Ce qu'il a obtenu :	une affectation de `b` à `a`, le résultat étant la valeur affectée.

Pourquoi tant de haîne ? L'affectation est un opérateur et non pas une instruction.

C.1.2 Erreur sur l'affectation

C'est l'inverse de l'erreur précédente, mais qui reste beaucoup moins fréquente.

Ce que voulait le programmeur :	Affecter `b` à `a`
Ce qu'il aurait dû écrire :	`a = b`
Ce qu'il a écrit :	`a == b`
Ce qu'il a obtenu :	La comparaison de `a` à `b`.

C.2 Erreurs avec les macros

Le mécanisme des macros est réalisé par le préprocesseur qui réalise un traitement en amont du compilateur proprement dit. Ce concept est abordé en détail dans le chapitre 7. Comme le traitement des macros est un pur traitement textuel, sans aucun contexte, c'est un nid à erreurs.

C.2.1 Un `#define` n'est pas une déclaration

Ce que voulait le programmeur :	la définition de la constante `MAX` initialisée a 10
Ce qu'il aurait dû écrire :	`#define MAX 10`
Ce qu'il a écrit :	`#define MAX 10 ;`

Cette erreur peut provoquer ou non une erreur de compilation à l'utilisation de la macro :
- L'utilisation `x = MAX ;` aura pour traduction `x = 10 ; ;`, ce qui est possible : il y a une instruction nulle derrière l'instruction `x = 10 ;`
- L'utilisation `int t[MAX] ;` aura pour expansion `int t[10 ;] ;` ce qui génèrera un message d'erreur.

C.2.2 Un `#define` n'est pas une initialisation

Ce que voulait le programmeur :	la définition de la constante `MAX` initialisée a 10
Ce qu'il aurait dû écrire :	`#define MAX 10`
Ce qu'il a écrit :	`#define MAX = 10 ;`

Cette erreur sera généralement détectée à la compilation, malheureusement le message d'erreur sera émis sur l'utilisation de la macro, et non pas là où réside l'erreur, à savoir la définition de la macro.

C.2.3 Erreur sur macro avec paramètres

La distinction entre macro avec paramètres et macro sans paramètre se fait sur la présence d'une parenthèse ouvrante juste après le nom de la macro, sans aucun blanc entre les deux. Ceci peut amener des résultats surprenant ; comparer les deux exemples suivants :

Définition de la macro	**paramètres**	**traduction**
`#define add(a,b) (a + b)`	`a` et `b`	`(a + b)`
`#define add (a,b) (a + b)`	aucun	`(a,b) (a + b)`

C.2.4 Erreur avec les effets de bord

Le corps d'une macro peut comporter plusieurs occurrences d'un paramètre. Si à l'utilisation de la macro on réalise un effet de bord sur le paramètre effectif, cet effet de bord sera réalisé plusieurs fois. Exemple :
`#define CARRE(a) ((a) * (a))`
l'utilisation de `CARRE(x++)` aura comme traduction `((x++) * (x++))` et l'opérateur `++` sera appliqué deux fois.

C.3 Erreurs avec l'instruction `if`

L'instruction `if` ne comporte ni mot-clé introducteur de la partie `then`, ni terminateur (pas de `fi` dans le style des `if then else fi`).
Ceci peut provoquer les erreurs suivantes :

Ce que voulait le programmeur :	tester si `a > b`
Ce qu'il aurait dû écrire :	`if ( a > b)` `  a = b;`
Ce qu'il a écrit :	`if ( a > b);` `  a = b;`

Le problème vient aussi du fait de l'existence de l'instruction nulle :

Ce que voulait le programmeur :	tester si `a > b`

Ce qu'il aurait dû écrire :	`if ( a > b)`
	`  { if ( x > y)  x = y; }`
	`else`
	`  ...`

Ce qu'il a écrit :	`if ( a > b)`
	`  if ( x > y)  x = y;`
	`else`
	`  ...`

On rappelle qu'un `else` est raccroché au premier `if`.

C.4 Erreurs avec les commentaires

Il y a deux erreurs classiques avec les commentaires :

1. le programmeur oublie la séquence fermante `/*`. Le compilateur "mange" donc tout le texte jusqu'à la séquence fermante du prochain commentaire.
2. On veut enlever (en le mettant en commentaire) un gros bloc d'instructions sans prendre garde au fait qu'il comporte des commentaires. Les commentaires ne pouvant être imbriqués, ça n'aura pas l'effet escompté par le programmeur. La méthode classique pour enlever (tout en le laissant dans le source) un ensemble d'instructions est d'utiliser le préprocesseur :

```
#ifdef NOTDEFINED
...
#endif
```

C.5 Erreurs avec les priorités des opérateurs

Les priorités des opérateurs sont parfois surprenantes. Les cas les plus gênants sont les suivants :

– La priorité des opérateurs bit à bit est inférieure à celle des opérateurs de comparaison.

Le programmeur a écrit	**il désirait**	**il a obtenu**
`x & 0xff == 0xac`	`(x & 0xff) == 0xac`	`x & (0xff == 0xac)`

– La priorité des opérateurs de décalage est inférieure à celle des opérateurs arithmétiques.

Le programmeur a écrit	**il désirait**	**il a obtenu**
`x << 4 + 0xf`	`(x << 4) + 0xf`	`x << (4 + 0xf)`

– La priorité de l'opérateur d'affectation est inférieure à celle des opérateurs de comparaison. Dans la séquence ci-dessous, très souvent utilisée, toutes les parenthèses sont nécessaire :
```
while ((c = getchar()) != EOF) {
    ...
    }
```

C.6 Erreur avec l'instruction `switch`

C.6.1 Oubli du `break`

Sans l'instruction `break`, le programme continuera à l'alternative suivante ce qui peut poser problème. (voir §2.2.5).

C.6.2 Erreur sur le `default`

L'alternative à exécuter par défaut est introduite par l'etiquette `default`. Si une faute de frappe est commise sur cette étiquette, l'alternative par défaut ne sera plus reconnue : l'étiquette sera prise pour une étiquette d'instruction sur laquelle ne sera fait aucun `goto` (voir §2.2.6).

```
switch(a) {
     case 1 : a = b;
     defult : return(1);       /*   erreur non détectée    */
}
```

Version diabolique de cette erreur : si la lettre l de default est remplacée par le chiffre 1, avec les fontes utilisées pour imprimer les sources, qui verra la différence entre l et 1 ?

C.7 Erreur sur les tableaux multidimensionnels

La référence à un tableau `t` à deux dimensions s'écrit `t[i][j]` et non pas `t[i,j]` comme dans d'autres langages de programmation.
Malheureusement, si on utilise par erreur la notation `t[i,j]` et selon le contexte d'utilisation, elle pourra être acceptée par le compilateur. En effet, dans cette expression la virgule est l'opérateur qui délivre comme résultat l'opérande droite après avoir évalué l'opérande gauche. Comme l'évaluation de l'opérande gauche ne réalise ici aucun effet de bord, cette évaluation est inutile , donc `t[i,j]` est équivalent à `t[j]` qui est l'adresse du sous-tableau correspondant à l'index `j`.

C.8 Erreur avec la compilation séparée

Une erreur classique est d'avoir un tableau défini dans une unité de compilation :
`int tab[10];`
et d'utiliser comme déclaration de référence dans une autre unité de compilation :

`extern int * tab ;`
Rappelons que `int tab[]` et `int *t` ne sont équivalents que dans le seul cas de paramètre formel de fonction.
Dans le cas qui nous occupe ici, la déclaration de référence correcte est :
`extern int tab[] ;`

C.9 Liens utiles

En plus des ouvrages cités dans la bibliographie, voici quelques sites web qui m'ont servis pour réaliser ce polycopié et/ou qui m'ont parus intéressants :

- [Cas98] : `http://www-clips.imag.fr/commun/bernard.cassagne/Introduction_ANSI_C.html`
- `http://fr.wikibooks.org/wiki/Programmation_C`
- [Can03] : `http://www-rocq.inria.fr/codes/Anne.Canteaut/COURS_C/`
- [Fab02] : `http://www.ltam.lu/Tutoriel_Ansi_C/` (un très bon tutorial en ligne).

Bibliographie

[Can03] Anne Canteaut. "Programmation en Langage C". INRIA - projet CODES, 2003. http://www-rocq.inria.fr/codes/Anne.Canteaut/COURS_C/.

[Cas98] Bernard Cassagne. "Introduction au Langage C". Laboratoire CLIPS UJF/CNRS, 1997-1998. http://www-clips.imag.fr/commun/\\bernard.cassagne/Introduction_ANSI_C.html.

[Fab02] F. Faber. Introduction à la programmation en ANSI-C. Technical report, Tutorial, Août 2002. http://www.ltam.lu/Tutoriel_Ansi_C/.

[Her01] Matthieu Herrb. Guide superflu de programmation en langage C. Technical report, CNRS/LAAS, Dec. 2001. http://www.laas.fr/~matthieu/cours/c-superflu/index.

[KR88] B.W. Kernighan and D.M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1988. 2nd edition.

[Pil04] Jean-François Pillou. Introduction au Langage C. Technical report, Tutorial, Encyclopédie Informatique Libre, 2004. http://www.commentcamarche.net/c/cintro.php3.

[PKP03] Peter Prinz and Ulla Kirch-Prinz. *C Pocket Reference*. O'Reilly & Associates, 2003.

[Ste90] David Stevenson. "IEEE Std 754-1985 IEEE Standard for Binary Floating-Point Arithmetic". Technical report, IEEE Standards Association, 1990.