1 Introduction

Ce cours présente le langage C, un langage essentiel dans l'enseignement de la programmation parce qu'il occupe une place prépondérante en informatique, qu'il possède la majorité des constructions qu'on retrouve dans les autres langages structurés modernes et que sa syntaxe a servi de base à nombre de ces langages. Ce cours est à l'informatique à peu près ce que l'algèbre et le calcul différentiel et intégral sont aux mathématiques. Les notions qu'il présente ne permettent de résoudre que des problèmes simples, mais elles sont fondamentales.

Le cours est composé d'une série d'exercices introduisant progressivement les instructions du C. Il n'est pas nécessaire de les faire tous systématiquement. En fonction de ses capacités, le lecteur peut en faire un plus ou moins grand nombre. Ces exercices sont là d'une part pour s'entraîner et pour mieux comprendre les instructions présentées, et d'autre part pour vérifier l'acquisition des connaissances.

Ce cours se termine par des mini-projets montrant à quoi sert un ordinateur dans l'industrie et la recherche:

- calcul de la trajectoire d'un projectile
- rotation 3-D
- calcul d'un pont en treillis
- labyrinthe

2 Description générale de l'ordinateur

Cette section décrit très brièvement l'architecture des ordinateurs et le codage de l'information dans un ordinateur. Ces informations sont utiles pour mieux comprendre l'organisation et le fonctionnement des programmes en C.

2.1 Architecture des ordinateurs

La Figure 1 montre les éléments principaux d'un ordinateur. Celui-ci comporte essentiellement une mémoire électronique et un processeur connectés entre eux et connectés aux périphériques (le disque dur, le clavier, l'écran, etc.). La mémoire électronique est rapide. Elle contient les données (valeurs numériques, textes, dessins...) et les programmes (listes d'instructions qui font précisément l'objet de ce cours). Quand on éteint l'ordinateur, cette mémoire s'efface, à part une petite partie qui permet de relancer l'ordinateur au réenclenchement. La mémoire électronique est trop petite pour contenir tout ce qu'on veut mémoriser. C'est pourquoi l'ordinateur a à sa disposition des mémoires plus grandes mais plus lentes: le disque dur et les disquettes. Ces mémoires-là retiennent les informations quand on éteint l'ordinateur.

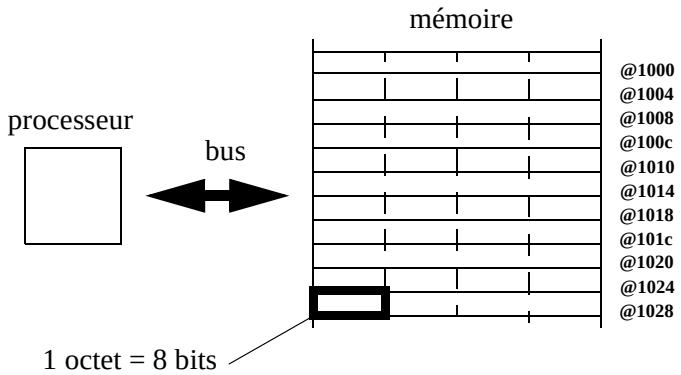


FIGURE 1: La structure d'un ordinateur

Le processeur possède une unité arithmétique et logique qui prend les instructions une à une dans la mémoire électronique et les exécute. A l'enclenchement l'ordinateur commence par exécuter des instructions à la case-mémoire numéro zéro et, jusqu'à ce qu'on l'éteigne, il n'arrête pas d'en exécuter et d'en réexécuter, chargeant sa mémoire avec des programmes en provenance du disque ou créés selon vos ordres.

Tout ce qui apparaît sur l'écran lorsque vous enclenchez l'ordinateur a été dessiné par des programmes, à peu près identiques sur chaque station de travail. Ces programmes détectent également les mouvements de la souris et les touches pressées sur le clavier. Ils manipulent les informations enregistrées sur les disques et les disquettes. Ils exécutent des éditeurs de textes ou graphiques, des calculateurs. De tels programmes peuvent être développés à l'aide d'autres programmes précisément prévus pour cela. Bien évidemment le tout premier programme a dû être développé d'une autre façon, mais c'était il y a longtemps.

2.2 Le codage de l'information

Tout fonctionne en binaire dans un ordinateur, c'est-à-dire en manipulant les seules valeurs 0 et 1, représentées habituellement dans la machine par les tensions de 0 et 5V respectivement. Ces chiffres binaires, 0 et 1, à partir desquels on construit des nombres plus grands sont appelés *bits*, abréviation de *binary digit*. Toutes les données manipulables par un ordinateur sont ainsi représentées par des séquences de bits:

- Un caractère: 8 bits (code entre 0 et 255)
- Un entier: 32 bits
- Un réel en virgule flottante (32 ou 64 bits).
- sons: décomposés en échantillons
- images: décomposées en pixels.

Pour comprendre le codage de l'information, il faut connaître les puissances de 2:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1

Codage d'un entier de 8 bits. La figure 2 montre comment est codé un entier de 8 bits. La valeur binaire. 01100001 correspond à l'entier 97. Pour savoir cela, on aligne au dessus de chaque bit l'exposant de 2 correspondant à la position du bit, et l'on additionne toutes les puissances de 2 pour lesquelles le bit est 1. Dans notre cas, cela donne 64+32+1 = 97. Il est facile d'étendre cela à 8, 16, 32 ou 64 bits. Les nombres de 64 bits permettent de stocker des valeurs astronomiques (2^{64} ~= 16.10^{18}), soit 16 milliards de milliards.

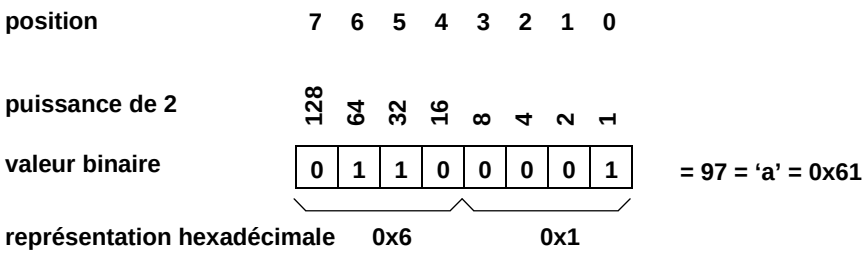


FIGURE 2: Codage binaire

Codage des caractères. Les caractères sont généralement représentés sur 8 bits. Par convention, on fait correspondre certaines valeurs aux caractères de l'alphabet. La convention de codage la plus fréquente est la convention ASCII, qui fait correspondre les valeurs 97, 98, 99, ... aux caractères 'a', 'b', 'c', ..., et les valeurs 65, 66, 67, ... aux caractères 'A', 'B', 'C', ... La figure 3 résume ce codage des caractères. Notez que les caractères

figurant en italique dans cette table ne sont pas affichables tels quels mais sont ce que l'on appelle des caractères de contrôle (retour à la ligne, changement de page, bip sonore, etc...)

0	*NU*	16	*DL*	32	*SP*	48	0	64	@	80	P	96	‘	112	p
1	*SH*	17	*D1*	33	!	49	1	65	A	81	Q	97	a	113	q
2	*SX*	18	*D2*	34	"	50	2	66	B	82	R	98	b	114	r
3	*EX*	19	*D3*	35	#	51	3	67	C	83	S	99	c	115	s
4	*ET*	20	*D4*	36	$	52	4	68	D	84	T	100	d	116	t
5	*EQ*	21	*NK*	37	%	53	5	69	E	85	U	101	e	117	u
6	*AK*	22	*SY*	38	&	54	6	70	F	86	V	102	f	118	v
7	*BL*	23	*EB*	39	’	55	7	71	G	87	W	103	g	119	w
8	*BS*	24	*CN*	40	(	56	8	72	H	88	X	104	h	120	x
9	*HT*	25	*EM*	41	)	57	9	73	I	89	Y	105	i	121	y
10	*LF*	26	*SB*	42	*	58	:	74	J	90	Z	106	j	122	z
11	*VT*	27	*EC*	43	+	59	;	75	K	91	[	107	k	123	{
12	*FF*	28	*FS*	44	,	60	<	76	L	92	\	108	l	124	|
13	*CR*	29	*GS*	45	-	61	=	77	M	93	]	109	m	125	}
14	*SO*	30	*RS*	46	.	62	>	78	N	94	^	110	n	126	~
15	*SI*	31	*US*	47	/	63	?	79	O	95	_	111	o	127	*DT*

FIGURE 3: Codage ASCII des caractères

Notation hexadécimale. Comme la notation binaire demande énormément de chiffres pour représenter un nombre et que la notation décimale usuelle ne permet pas de manipuler les puissances de 2 facilement, on utilise souvent la notation hexadécimale, c'est-à-dire en base 16. Pour convertir un nombre noté en binaire en notation hexadécimale, il suffit de grouper les bits par blocs de 4, correspondant à des valeurs entre 0 et 15, notées 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a (= 10), b (= 11), c (= 12), d (= 13), e (= 14), f (= 15). On peut ainsi représenter la valeur décimale 97, dans la figure 2, par la valeur hexadécimale 0x61. Par convention et pour éviter les confusions, on fait toujours commencer les valeurs hexadécimales par les caractères 0x.

Les codages des nombres réels, des sons et des images sont un peu plus compliqués, mais se réduisent toujours à des séquences de nombres entiers manipulés par l'ordinateur sous forme de chaînes de bits.

2.3 Fonctionnement d'un processeur

Au bas niveau (niveau matériel), la seule chose qu'un microprocesseur sait faire, c'est prendre une ou deux valeurs en mémoire et les stocker temporairement dans un *registre* interne (un registre est une petite mémoire très rapide interne au microprocesseur), leur faire subir des opérations simples (et logique, ou logique, addition, soustraction), et remettre le résultat en mémoire. Ces trois fonctions (prendre les valeurs en mémoire, opération simples, et remettre les valeurs en mémoire) s'appellent les *instructions* du processeur. Toutes les opérations complexes que vous ferez avec votre ordinateur se décomposent en une séquence (parfois très longue) d'instructions.

Les microprocesseurs modernes effectuent environ 1 instruction par cycle d'horloge. Lorsque l'on sait que les microprocesseurs actuellement sur le marché fonctionnent à 500Mhz (et certains à 1Ghz), cela laisse au microprocesseur le temps d'effectuer 500 millions d'instructions par seconde.

Chaque type d'instruction a un code (par exemple 0 pour charger une valeur de la mémoire dans un registre du processeur, 1 pour le mouvement inverse, 2 pour l'addition de deux valeurs contenues dans un registre, etc.). Un programme simple pourrait donc être la séquence de chiffres suivante:

0-0-@1000	charger dans le registre 0 le contenu de la mémoire à l'adresse 1000

Tableau 1: Un programme simple en langage machine

0-1-@1004	charger dans le registre 1 le contenu de la mémoire à l'adresse 1004
2-0-1-5	additionner le contenu des registres 0 et 1 et mettre le résultat dans le registre 5
1-5-@1008	décharger le contenu du registre 0 dans la mémoire à l'adresse 1008

Tableau 1: Un programme simple en langage machine

Dans le tableau ci-dessus, la première colonne contient la valeur chiffrée de chaque instruction, et la deuxième le comportement correspondant. Dans chaque instruction, le premier chiffre est le type d'instruction (chargement, déchargement, addition, ...), et les deux autres chiffres sont des numéros de registre ou des adresses mémoire. Par exemple, la première instruction (0-0-@1000) veut dire: charger - dans le registre 0 - le contenu de la mémoire à l'adresse 1000. C'est ce que l'on appelle le langage machine.

2.4 La programmation

Programmer, c'est écrire une suite de chiffre semblable à celle de la table 1. Ecrire des suites de chiffres comme cela est excessivement difficile. Les relire et les corriger est virtuellement impossible. En plus, la séquence de chiffres requise pour faire la même opération sur deux ordinateurs différents (PC, Mac, station) n'est pas la même. C'est pourquoi on a dû inventer des langages de haut niveau, que l'on *compile*, c'est-à-dire que l'on transforme en langage machine. L'instruction correspondant à l'instruction ci-dessus est par exemple:

```
NouveauCapital = AncienCapital + Interet ;
```

Programme 1

La correspondance entre ce programme et la table 1 est la suivante: les cases mémoire @1000, @1004 et @1008 ont reçu les noms de `AncienCapital`, `Interet` et `NouveauCapital`. Et le programme demande d'additionner l'`AncienCapital` et l'`Interet` et de déposer le résultat dans la case mémoire associée au nom `NouveauCapital`.

Le programme 1 est certainement beaucoup plus lisible: cela provient du fait que l'on a donné un nom en langage courant aux emplacements mémoire, et utilisé le signe '`+`' conventionnel pour l'addition. Le signe '`=`' veut simplement dire: 'copier le résultat à l'emplacement mémoire désigné par le nom `NouveauCapital`.

Le *compilateur* se charge de transformer le programme 1 en la séquence d'instruction du tableau 1, en choisissant lui-même un bon endroit dans la mémoire, et en choisissant les bons codes d'instructions. Les compilateurs modernes peuvent aider énormément le programmeur dans sa tâche.

2.5 Un peu d'histoire...

Le langage C est un langage déclaratif compilé conçu pour être très efficace et facilement portable d'un ordinateur à l'autre. Ce langage a été mis au point par Brian Kernighan et Dennis Ritchie des Bell Laboratories en 1972. C'est un langage structuré offrant un niveau d'abstraction relativement faible par rapport aux données et opérations réellement manipulées par la plupart des microprocesseurs, ce qui permet d'assurer une grande rapidité d'exécution. C'est pourquoi le langage C est le langage de prédilection pour le développement des systèmes d'exploitation, d'ailleurs le langage C lui-même a été développé pour faciliter le portage du système d'exploitation UNIX sur diverses architectures matérielles. Le langage C est probablement le langage le plus utilisé par les professionnels de la programmation de nos jours, parce qu'il allie les avantages d'un langage de plus haut niveau à ceux de l'assembleur[1]. De plus, la définition du langage C est du domaine public. Les compilateurs commercialisées actuellement par des éditeurs de logiciels sont néanmoins protégées par des droits d'auteur. C'est à la fin de 1983, que Microsoft et Digital Research ont publié le premier compilateur C pour

1. L'assembleur est un langage symbolique permettant de manipuler directement les registres et instructions d'un microprocesseur. L'assembleur a donc une syntaxe spécifique à chaque microprocesseur et qui varie beaucoup d'un microprocesseur à l'autre.

micro-ordinateur personnel. L'institut américain de normalisation, ANSI, a ensuite normalisé ce langage en 1989.

D'une syntaxe simple, le langage C a également servi de base à de nombreux langages dérivés plus modernes tels le langage C++ dont la première version date de 1983 ou Objective C qui sont tous deux des extensions orientées objet compatibles avec le langage C. Le langage Java, lui aussi, emprunte l'essentiel de sa syntaxe au langage C. C demeure donc un préalable quasiment incontournable pour qui s'intéresse à la plupart des autres langages plus récents.

3 Une première session à votre station de travail

Voir polycopié donné au premier cours.

4 L’environnement de programmation

Voir polycopié donné au premier cours.

5 Quelques programmes C simples

5.1 Comment fait-on pour écrire sur l’écran?

Ci-dessous on voit le texte du premier exercice. Il écrit “*Bonjour Hal*” puis “*Belle journée*” sur la fenêtre de texte.

```
#include <stdio.h>
void main ()
{
  printf("Bonjour Hal\n");
  printf("Belle journée");
}
```

Programme 2

Ces quelques lignes forment un programme, qui contient les éléments essentiels qui se retrouvent dans tout programme C. Nous allons donc survoler ces différents éléments et nous les détaillerons dans les sections suivantes.

Un programme C est composé de *fonctions* et de *variables*. Une fonction est elle-même constituée d'une séquence d'*instructions* qui indiquent les opérations à effectuer alors que les variables mémorisent les valeurs utilisées au cours du traitement. Les instructions sont elles-mêmes constituées de *mots-clés*, de *variables* et de *fonctions*. Les *mots-clés*, représentés ici en gras[1], sont les instructions de base du langage qui permettent au programmeur d’indiquer la structure et le déroulement de son programme. Ces mots-clés sont propres au langage. Un autre langage (Basic, Pascal, ADA) aura d’autres mots-clés.

Les noms donnés aux variables et fonctions sont ce qu’on appelle des *identificateurs*. Ils doivent commencer par une lettre (majuscule ou minuscule non accentuée) mais peuvent contenir outre des lettres, des chiffres et le caractère souligné _ dans le reste du symbole. En langage C, les caractères majuscules et minuscules ne sont pas équivalents, ainsi `printf()` et `PRINTF()` désignent deux fonctions différentes. En règle générale toutefois, on évite d'utiliser des identificateurs ne différant que par leur casse.

1. Les mots-clés ont été mis en gras ici uniquement pour faciliter la lecture du programme. La mise en gras n'a pas de signification pour le compilateur qui ne comprend que le texte brut sans aucun attribut de présentation (gras, italiques, etc...)

La première ligne contient la directive **`#include`** suivi d'un nom de fichier, ici `stdio.h`, contenant les déclarations nécessaires à l'utilisation de certaines fonctions. Le compilateur dispose ainsi des informations nécessaires pour vérifier si l'appel de la fonction (en l'occurrence `printf`) est correct.

A la deuxième ligne la construction **`main()`** indique le début de définition d'une *fonction* dénommée `main`. Tout programme C comporte au minimum une fonction, la fonction **`main`**, qui est la fonction principale par laquelle commence l'exécution du programme.

Les accolades `{` et `}` délimitent un *bloc* d'instructions constituant le corps de la fonction **`main`**, en l'occurrence, deux instructions `printf`. La fin de chaque instruction est marquée par un point-virgule `;`.

Les mots entre guillemets constituent ce que l'on appelle une *chaîne de caractères* (*character string*), et sont imprimés tels quels par l'instruction `printf` à quelques exceptions près (voir les section 5.2 et 6.3).

Lorsqu'on exécute le programme, comme vous l'avez certainement fait dans le programme d'introduction, les instructions `printf` affichent le texte qui se trouve entre les guillemets. L'instruction `printf("Bonjour Hal\n")` affiche donc "Bonjour Hal" dans la fenêtre texte.

5.2 L'instruction printf

La fonction `printf` ne fait pas partie du langage C au sens strict mais d'une bibliothèque de fonctions standard d'entrées/sorties, *stdio*, abréviation de *Standard Input/Output*, qui sont toujours fournies avec le langage, c'est ce qui explique la présence de la directive `#include <stdio.h>` en début de programme. La fonction `printf` ne provoque pas de retour à la ligne après la chaîne de caractères qu'elle affiche. Pour provoquer un retour à la ligne il faut insérer séquence spéciale `\n` dans la chaîne de caractères.

```
#include <stdio.h>
void main ()
{
  printf("Un ");
  printf("bout de ");
  printf("texte.\n");
  printf("Nouvelle ligne");
}
```

Programme 3

Ces instructions écrivent:

```
Un bout de texte.
Nouvelle ligne
```

Ecran 1

La séquence `\n` représentant le caractère de saut de ligne est appelée *séquence d'échappement*. Les séquences d'échappement permettent de représenter des caractères non imprimables ou difficiles à taper. Outre `\n` on trouve ainsi `\t` pour le caractère de tabulation (*tab*), `\b` pour le caractère de retour en arrière (*backspace*), `\"` pour le guillemet et `\\` pour le caractère `\` lui-même. Ainsi, si l'on veut afficher un texte qui contient un guillemet, on ne peut pas simplement le mettre dans le texte, car il sert également à indiquer la fin de la chaîne, on doit utiliser la séquence d'échappement correspondante:

```
printf("Le guillemet \" delimite les chaines");
```

Exercice 1. *Créer un programme qui affiche dans la fenêtre texte:*

```
Bonjour !
     Bonjour !
          Bonjour !
```

Ecran 2

Pour décaler les mots, utilisez des espaces.

5.3 Commentaires

Il est indispensable de mettre des commentaires dans les programmes d'une certaine envergure car il est très difficile de retrouver ce que fait un programme quand on n'a que la liste d'instructions. En C, il est possible de mettre des textes entre les séquences `/*` et `*/` ou après la séquence `//`[1], pour documenter les programmes directement dans les sources. Ces commentaires ne font pas partie des instructions et sont totalement ignorés par le compilateur.

```
/* Commentaire
   qui s'étend sur
   plusieurs lignes */

// Commentaire sur une ligne
```

6 C plus en détail

6.1 Comment mémorise-t-on une valeur?

Les *variables* sont des espaces de mémoire qui permettent de conserver des nombres ou d'autres éléments (lettres, mots...). Chaque variable est caractérisée par son nom, son *type* et son contenu. Le nom est un identificateur et doit donc commencer par une lettre, mais peut contenir outre des lettres, des chiffres et le caractère souligné _ dans le reste du symbole. Les variables correspondent plus ou moins à celles qu'on utilise en algèbre, mais le contenu des variables de programme peut être modifié en cours de route. Ne pas confondre les inconnues d'une équation d'algèbre avec une variable de programme. Une variable est en fait un simple récipient dont on peut changer ou lire le contenu.

Une variable qui permet de "compter" est une variable entière. Le programme ci-dessous montre comment déclarer des variables et comment déposer des valeurs dans celles-ci.

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

  un_nombre = 5;
}
```

Programme 4

Pour pouvoir manipuler une variable il est nécessaire d'annoncer cette variable au préalable. Cette opération s'appelle une *déclaration*. Ainsi à la première ligne du corps de ce programme, après l'accolade `{` on trouve le mot-clé **`int`**, qui indique que l'on souhaite déclarer une variable de type entier (*integer*), suivi du nom donné à cette nouvelle variable.

Toute variable utilisée dans un programme doit avoir été déclarée auparavant.

On peut ainsi déclarer autant de variables que l'on désire. Les déclarations doivent toujours être les premières instructions au début d'un bloc marqué par une accolade ouvrante {. Aucune déclaration ne peut intervenir après une instruction autre qu'une déclaration.

1. La séquence // a été introduite par le langage C++ et n'est pas supportée par les compilateurs C stricts. En cas de doute il vaut mieux utiliser les séquence /* */ même pour des commentaires d'une seule ligne

Immédiatement après sa déclaration le contenu d'une variable est indéterminé, dépendant de l'ordinateur sur lequel le programme est exécuté. Lors de l'exécution du programme, on peut y déposer des valeurs provenant de calculs ou de lectures au clavier. Ainsi, l'instruction `un_nombre = 5;` dépose la valeur 5 dans la variable. Cette instruction s'appelle *affectation*.

Pour éviter qu'une variable ne prenne une valeur indéterminée jusqu'à sa première affectation, on peut spécifier sa valeur au moment de sa déclaration. Cette construction s'appelle une *initialisation*:

```
int un_nombre = 5;
```

Le type entier associé au mot-clé `int`, signifie qu'une variable de ce type ne peut contenir que des nombres entiers. Les valeurs extrêmales acceptables pour un tel entier dépendent de l'architecture (ordinateur et système d'exploitation) utilisée. En règle générale, actuellement, les entiers ont une taille de 32 bits ce qui autorise des valeurs de -2'147'483'648 à 2'147'483'647. Pour manipuler des nombres réels, non entiers, on utilise les types **`float`**, qui signifie nombre à virgule flottante ou plus simplement flottant, et **`double`**, qui signifie flottant à double précision. Un nombre à virgule flottante est une quantité codée sur 32 bits, comprenant au moins six chiffres significatifs, et comprise entre 10^{-38} et 10^{+38} environ. Une variable de type **`float`** ou **`double`** peut ainsi contenir par exemple des valeurs très proches de (mais jamais exactement égales à) Π ou $\sqrt{2}$. Un flottant peut posséder des chiffres après la virgule, la virgule étant représentée par un point conformément à l'usage des pays anglo-saxons. Une variable de type flottant, **`float`**, et une variable de type entier, **`int`**, ont été déclarées dans le programme ci-dessous:

```
#include <stdio.h>
void main ()
{
  float nombre_reel;
  int lon;

  nombre_reel = 5.679 + 6.120;
  lon = 31645;
}
```

Programme 5

S'il y a plusieurs variables de même type, elles peuvent être déclarées sur la même ligne, séparées par des virgules comme on le voit dans cette déclaration: `int i, j, m1, m2, m3, m4, m5;`

L'ordinateur n'annonce en général pas les dépassements de capacité. Si une variable `nombre` a été déclarée de type `int` et que l'on effectue un calcul aboutissant à une valeur plus grande que la valeur maximale autorisée: `nombre = 1111111 * 2222222;` le résultat n'est pas correct car le résultat est plus grand que 2^{31}. `1111111` et `2222222` sont bien des entiers mais leur multiplication aboutit à une valeur qui n'est pas représentable par un entier. Cependant à part des cas rares, les entiers, `int`, suffisent et le problème évoqué ci-dessus n'apparaît pas souvent.

Il existe d'autres types numériques. Ainsi les types **`long int`** et **`short int`**, que l'on peut abréger en **`long`** ou **`short`**, représentent des entiers tout comme **`int`** mais avec des valeurs minimales et maximales qui peuvent être différentes. En principe un **`int`** est un entier représenté dans la taille la plus naturelle pour le microprocesseur utilisé, c'est-à-dire 32 bits sur un microprocesseur 32 bits et 64 bits sur un microprocesseur 64 bits mais ceci dépend aussi du système d'exploitation voire du compilateur (on peut avoir des **`int`** de 32 bits sur une machine 64 bits). Dans tous les cas un **`short`** fait au moins 16 bits, un **`long`** au moins 32. Le fichier `limits.h` définit les constantes symboliques `INT_MIN`, `INT_MAX`, `SHRT_MIN`, `SHRT_MAX`, `LONG_MIN`, `LONG_MAX` qui indiquent les bornes de chacun de ces types.

On peut également appliquer les qualificatifs **`signed`** et **`unsigned`** aux types **`char`**, **`int`**, **`short`** et **`long`**. Ces qualificatifs indiquent si le type est signé ou non signé ce qui change l'arithmétique et les valeurs maximales autorisées pour une variable du type considéré. Ainsi, une variable de type **`signed char`** peut contenir des valeurs allant de -128 à +127 alors qu'une variable de type **`unsigned char`** peut contenir des valeurs allant de 0 à 255. Par défaut les types sont signés et le mot-clé **`signed`** est donc très peu utilisé en pratique, le mot-clé **`unsigned`** est par contre très fréquent. Là encore le fichier `limits.h` contient des constantes définissant les valeurs maximales admises pour les types non signés: `UCHAR_MAX`, `UINT_MAX`, `USHRT_MAX` et `ULONG_MAX`

On prend rarement assez de précautions quand on manipule un mélange de variables signeés et non signées. Comparer par exemple une variable signée avec une variable non signée peut ainsi donner des résultats surprenants. La plupart du temps le compilateur émet des avertissements à ce sujet.

6.2 Constantes

Les constantes entières tapées dans un programme telles que `125` sont par défaut de type `int`. On peut toutefois demander qu'elles soient considérées de type `long` en ajoutant un `l` ou `L` à la fin: `125L`. Une constante entière trop grande pour tenir dans `int` est considérée comme un `long` même si elle n'a pas de `l` ou `L` à la fin. Pour désigner une constante non signée on utilise le suffixe `u` ou `U` que l'on peut combiner avec le suffixe `l` ou `L` pour désigner une constante de type `unsigned long`: `1245UL`

On peut écrire les constantes entières en hexadécimal en les préfixant par `0x` ou `0X` (zéro X) comme dans `0X12AB`. On peut également les écrire en octal (base huit) en les préfixant par un zéro comme dans `0755`.

Les constantes contenant une virgule (`125.7`) ou un exposant (`1e2`) sont considérées de type `double` par défaut. On peut cependant demander que de telles constantes soient considérées soit comme un `float` en ajoutant un suffixe `f` ou `F`, soit comme un `long double` en ajoutant un `l` ou `L`.

Une constante de type caractère (`char`) s'écrit sous forme d'un caractère entre apostrophes: `'a'`. La valeur numérique d'une telle constante est la valeur du caractère dans le jeu de caractères de la machine (le plus souvent le code ASCII). Ainsi la valeur numérique de la constante caractère `'0'` est 48. Dans les calculs, les caractères sont traités exactement comme des entiers même si on s'en sert le plus souvent pour les comparer à d'autres caractères.

Attention à ne pas confondre `'a'` qui désigne la constante caractère `a` ayant pour valeur numérique 97 et `"a"` qui désigne un tableau de caractères contenant le caractère `a` suivi d'un caractère `'\0'` (voir section 6.12)

Constantes symboliques. Dans de nombreux programmes, il est agréable et plus clair d'utiliser une constante sous son nom habituel (pi, g...). Pour cela on utilise la directive **`#define`**, généralement à l'extérieur du bloc de la fonction `main`:

```
#include <stdio.h>

#define pi 3.1415626
#define g 9.81

void main ()
{
  double rayon, diametre;
  rayon = 15.0;
  diametre = 2*pi*rayon;
}
```

Programme 6

Dans le programme, les constantes s'utilisent comme les variables, à part le fait qu'elles ne peuvent évidemment pas apparaître dans le membre de gauche d'une affectation.

Constantes énumérées. Une énumération est une suite de valeurs entières constantes auxquelles on donne des noms symboliques. Par exemple:

```
enum fruits { pomme, poire, banane };
```

Le premier nom d'une énumération vaut zéro, le suivant un, et ainsi de suite, à moins que l'on précise des valeurs explicites. Si l'on ne donne que certaines valeurs, les suivantes se déduisent par incréments successifs de un:

```
enum mois { jan=1,fev,mar,avr,mai,jun,jul,aou,sep,oct,nov,dec};
```

Dans cet exemple `fev` vaut 2, `mar` vaut 3 et ainsi de suite.

Les noms définis à l'intérieur d'une énumération doivent être distincts.

6.3 Affichage des variables

Pour afficher le contenu de variables on utilise l'instruction `printf`. Jusqu'à présent nous avons utilisé cette fonction avec comme seul paramètre une simple chaîne de caractères. Mais la fonction `printf` permet également d'afficher le contenu de variables. Pour cela, la fonction a besoin de savoir quel est le type de ces variables (entier, flottant, simple caractère, etc...) et quel format d'affichage utiliser (combien afficher de chiffres après la virgule, utiliser la notation décimale habituelle ou la notation scientifique en puissances de 10, compléter les nombres trop petits par des espaces ou des zéros à gauche, etc...) Ces deux informations, type des variables à afficher et format d'affichage sont indiquées au moyen de séquences de caractères spéciales, appelées *spécificateurs de format*, insérées dans la chaîne de caractères passée comme premier paramètre à la fonction `printf`. Cette chaîne, dite chaîne de format (*format string*), est suivie du nom des variables à afficher, séparées par des virgules.

Ainsi l'instruction `printf("%d",un_entier);` affiche le contenu de la variable `un_entier` sur l'écran. Le spécificateur de format `%d`, placée dans la chaîne de format indique que la variable `un_entier` est de type `int`. Le tableau 2 suivant résume quelques uns des spécificateurs de format les plus courants reconnus par la fonction `printf`.

Format	Type de variable correspondant
`%c`	Caractère de type `char`
`%d` ou `%i`	Entier de type `int`
`%x`	Entier de type `int` affiché en notation hexadécimale
`%u`	Entier non signé de type `unsigned int`
`%f`	Nombre à virgule flottante de type `float` ou `double`
`%e` ou `%E`	Nombre à virgule flottante affiché en notation exponentielle
`%s`	Chaîne de caractères

Tableau 2: Spécificateurs usuels de format de la fonction printf

Il est également possible d'afficher le contenu de plusieurs variables avec la même instruction `printf` en insérant pour chacune un spécificateur dans la chaîne de format et en séparant les différentes variables par des virgules dans la chaîne de paramètres. Par ailleurs il est également possible d'insérer du texte à afficher autour des spécificateurs de format. Ainsi, si les variables entières `nbr_pieces` et `sorte` contiennent respectivement les valeurs 3123 et 534, l'instruction

```
printf("Nombre de pièces=%d sorte=%d", nbr_pieces, sorte);
```

affiche

```
Nombre de pièces=3123 sorte=534
```

Exercice 2. *Faire un programme qui déclare la variable* `ma_valeur` *de type entier, qui lui affecte la valeur* `111*222` *puis qui affiche cette variable avec son nom.*

Contrôle de la longueur de champ d'affichage. Pour améliorer la lisibilité de nombres de longueur variable affichés en colonne, la fonction `printf` permet d'ajouter aux spécificateurs de format un nombre indiquant la largeur minimale de champ d'affichage, les nombres sont alors affichés alignés à droite dans des champs de longueur correspondante. Ainsi si l'on place un nombre au milieu du spécificateur `%d` dans la chaîne de format de l'instruction `printf`, comme suit:

```
printf("%6d", un_nombre);
```

la variable un_nombre sera affichée dans un champ de 6 caractères. Si la valeur contenue un_nombre comporte, par exemple, 4 chiffres alors deux espaces seront insérées avant d'afficher un_nombre. Si, par contre, un_nombre comporte 6 chiffres ou plus alors aucune espace supplémentaire n'est inséré.

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

  nbr = 12; printf("%4d\n", nbr);
  nbr = 1234; printf("%4d\n", nbr);
  nbr = 31645; printf("%4d\n", nbr);
}
```

Programme 7

```
  12
1234
31645
```

Ecran 3

La largeur minimale de champ s'applique aussi aux variables de type float ou double. De plus, pour celles-ci, il est également possible d'indiquer une précision, c'est-à-dire combien de chiffres après la virgule doivent être affichés (sans indication de précision, 6 décimales sont affichées). La précision est indiquée à la suite de la largeur de champ, séparée par un point.

Ainsi l'instruction printf(“%12.3f”,x) affiche la valeur de la variable flottante x avec 3 décimales au plus dans un champ de 12 caractères minimum complété à gauche par des espaces si besoin.

Exercice 3. *Affichez 12345678901234567890 pour avoir un repère pour compter la longueur des champs (mettez cette liste entre guillemets!), mettez la valeur 156.0 * 135.0 dans une variable et affichez-la sous différentes formes. Produisez ainsi le schéma ci-dessous.*

```
12345678901234567890
    21060.000
variableX = 21060
```

Ecran 4

Sur la dernière ligne, le nombre n’a simplement pas de chiffres après la virgule. Essayez également la forme libre, sans mention de largeur de champ ou de précision.

6.4 Lecture du clavier

Il est possible de demander au programme de s’arrêter à un endroit de son exécution et d’attendre qu’on lui donne une valeur au clavier. L’instruction qui fait cela, nommée scanf, a le même effet qu’une affectation. Tout se passe comme si l’on pouvait écrire: un_nombre = “valeur tapée au clavier”. scanf est un peu le symétrique de printf et s'utilise de façon assez similaire. Ainsi, le premier argument passé à scanf doit être

une chaîne de format indiquant le type des paramètres suivants. Cette chaîne est suivie d'une liste de variables dont le contenu sera affecté d'après ce que l'utilisateur du programme aura tapé au clavier.

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

  printf("Donnez un nombre: ");
  scanf("%d", &un_nombre);
}
```

Programme 8

Lors de l'exécution du programme ci-dessus, l'ordinateur affichera "Donnez un nombre:" puis, lorsqu'il arrivera à l'instruction `scanf("%d", &un_nombre)`, il attendra que l'utilisateur tape une valeur entière au clavier. Cette valeur sera déposée dans la variable `un_nombre`. On peut mettre plusieurs variables dans l'instruction `scanf`. Lorsqu'on les tape au clavier, il faut les séparer par des blancs ou des retours de ligne.

Attention de ne pas oublier le signe `&` devant le nom des variables numériques passées en argument. La signification de ce signe sera expliquée à la section 8.

Exercice 4. *Lire une variable réelle du clavier et l'afficher. Voir la réaction du programme lorsqu'on lui fournit un entier. Lire une variable entière et lui fournir un réel.*

6.5 Comment écrit-on une expression arithmétique?

Les opérateurs arithmétiques du langage C sont les suivants: **`+`**, **`-`**, **`*`**, **`/`**, **`%`**, représentant respectivement l'addition, la soustraction (ou le changement de signe: `-x`), la multiplication, la division et finalement le reste de la division entière (modulo).

Les quatre opérateurs de base opèrent sur des entiers aussi bien que des réels. Si les opérandes sont tous deux entiers alors l'opération a lieu en arithmétique entière. Si l'un des deux opérandes au moins est réel, alors l'opération a lieu en arithmétique réelle. Il faut dès lors prendre garde au fait qu'un nombre écrit dans un programme est par défaut considéré entier s'il n'a pas de point décimal, ce qui peut donner des résultats surprenants dans le cas de certaines divisions comme le montre le tableau 3. De plus, si l'on affecte à une variable entière un nombre réel (ou une expression, telle une division de réels, dont le résultat est un réel), alors le compilateur effectue automatiquement l'arrondissement[1]. Par exemple si `i` est une variable entière, l'expression `i=100.0/45.0;` vaut `2`.

Conversion explicite de type. Comme nous venons de l'indiquer, le compilateur opère des conversions de type automatiques dans certaines expressions arithmétiques. Il est également possible au programmeur de demander ce type de conversions explicitement, cette construction est appelée *cast* et prend la forme suivante: par exemple si `i` est un entier alors `(float)i` correspond à la valeur de `i` mais en tant que nombre réel.

instruction	signification
`i = 10 / 3;`	`i` de type **`int`** reçoit la valeur 3, résultat de la division entière
`i = 10.0 / 3;`	`i` de type **`int`** reçoit la valeur 3 par arrondi automatique de la division réelle
`x = 10 / 3;`	`x` de type **`float`** reçoit la valeur 3, résultat de la division entière

Tableau 3: Arithmétique entière et arithmétique réelle

1. Un avertissement peut toutefois être généré par certains compilateurs si aucune conversion explicite de type n'est effectuée.

instruction	signification
`x = 10/3 * 3;`	`x` reçoit la valeur 9.0, le calcul est fait en entiers puis transformé en réel.
`x = 10.0/3 * 3;`	`x` reçoit la valeur 10.0 la division et la multiplication étant effectués en réels.
`x=i/j;`	Si les entiers `i` et `j` valent 7 et 2, `x` de type **`float`** reçoit la valeur 3, résultat de la division entière
`x=(float)i/j;`	Si les entiers `i` et `j` valent 7 et 2, `x` de type **`float`** reçoit la valeur 3.5, résultat de la division réelle de `i`, préalablement converti explicitement en réel, par `j` automatiquement converti en réel

Tableau 3: Arithmétique entière et arithmétique réelle

Le programme ci-dessous illustre l'emploi des opérateurs sur les nombres entiers.

```
#include <stdio.h>
void main ()
{
  int i, j, m1, m2, m3, m4, m5;

  printf("Donnez deux nombres: ");
  scanf("%d %d", &i, &j);
  m1 = i + j;
  m2 = i - j;
  m3 = i * j;
  m4 = i / j;
  m5 = i % j;
  printf("i + j = %8d\n", m1);
  printf("i - j = %8d\n", m2);
  printf("i * j = %8d\n", m3);
  printf("%1d = %1d * %1d + %1d\n", i, j, m4, m5);
}
```

Programme 9

Ce programme demande deux nombres au clavier qu'il dépose dans les variables `i` et `j`. Ensuite il dépose dans les variables `m1` à `m4` la somme, la différence, le produit et le quotient de ces deux nombres. Dans `m5` on dépose le reste de la division de `i` par `j`. La dernière ligne écrit donc toujours quelque chose de correct.[1]

Dans les expressions mathématiques, les parenthèses obéissent aux mêmes règles qu'en algèbre. L'instruction `y = 2 * z1 + 5 * z2;` a la même signification que `y = (z1 * 2) + (z2 * 5);`. Les `*` sont obligatoires, on ne peut pas écrire `y = 2 z1 + 5 z2`, comme en algèbre.

Comme nous l'avons déjà indiqué, une variable peut recevoir plusieurs valeurs de suite au cours de l'exécution du même programme. On peut même changer sa valeur à partir d'un calcul qui la contient:

```
i = 7;   i = i + 5;
```

Cette dernière affectation (attention il s'agit bien d'une affectation et non d'une égalité mathématique) modifie la première valeur de `i` et réécrit `12` par-dessus le `7`.

Arrondissements. Les deux fonctions suivantes ne font pas partie du langage C mais se trouvent dans une bibliothèque de fonctions standard. Pour pouvoir les utiliser, il faut employer la directive `#include <math.h>` en début du programme.

`ceil(x)`: renvoie le plus petit entier supérieur ou égal à l'argument réel `x`;

1. En fait les opérateurs de division entière et modulo % ne correspondent à la définition mathématique rigoureuse que pour les entiers positifs. Le fait que le résultat de cette expression soit toujours juste est dû au fait que dans le cas où un des deux opérandes est négatif, les erreurs commises sur le modulo et la division entière se compensent!

`floor(x)`: renvoie le plus grand entier inférieur ou égal à l'argument réel `x`;

Il est également possible de réaliser des arrondissements au moyen de l'opérateur de conversion de type explicite (*cast*). Si `x` est une variable réelle, `(int)x` est l'entier obtenu en supprimant la partie fractionnaire de `x`[1]. Le programme 10 donne des exemples d'arrondissements utilisant des casts.

```
#include <stdio.h>
void main ()
{
  float x;
  int entier;

  x = 3.54;
  entier = x;            // Conversion automatique en entier
  entier = (int)(5 + x); // Conversion explicite en entier
  entier = (int)(x+0.5); // Ajouter 0.5 donne un arrondi de x plutot qu'une troncature
}
```

Programme 10

Les fonctions mathématiques suivantes sont disponibles en C (parmi d'autres), elles sont déclarées, comme `ceil` et `floor`, dans le fichier `math.h`:

syntaxe	fonction
`sin(x)`	sinus en radians
`cos(x)`	cosinus en radians
`arctan(x)`	arc tangente en radians
`pow(x,y)`	`x` élevée à la puissance `y`
`sqrt(x)`	racine carrée
`abs(x)`	valeur absolue
`log(x)`	logarithme naturel

Tableau 4: Fonctions mathématiques usuelles

On peut donc créer une expression du genre: `y = cos(sqrt(y+5.0)) + abs((arctan(z));`

Compter le temps qui passe. L'instruction `x=time(NULL);` affecte à la variable `x` de type `time_t` (un entier long), le nombre de secondes écoulées depuis le 1er janvier 1970. Cette fonction permet par exemple de mesurer des intervalles de temps. Il suffit pour cela de l'appeler au début et à la fin de l'intervalle avec deux variables différentes et de faire la différence entre les deux valeurs ainsi obtenues. Cette fonction est déclarée dans le fichier `time.h`

Tirer des valeurs aléatoires. Pour déposer une valeur aléatoire dans une variable `i`, de type entier, il suffit d'écrire `i = rand();`. La fonction `rand` retourne une valeur entière comprise entre 0 et `RAND_MAX`. `RAND_MAX` dépend de l'architecture mais vaut au minimum 32767. Pour avoir une valeur réelle comprise entre `0.0` et `1.0`, il suffit de d'utiliser cette instruction: `x = (float)rand()/RAND_MAX;`. Chaque fois qu'on appelle la fonction `rand` dans la même exécution du programme, elle renvoie une nouvelle valeur aléatoire.

Toutefois le générateur aléatoire produit toujours la même séquence de nombres lorsque l'on ré exécute un programme, ce qui peut faciliter le dépannage des programmes mais peut également être parfois gênant. Pour éviter ce problème, il faut initialiser l'origine du générateur au moyen de la fonction `srand` avec un nombre

1. Ceci correspond au sens mathématique de la partie entière du nombre s'il est positif, à la partie entière plus un s'il est négatif, ce qui justifie l'existence de la fonction `floor`.

changeant à chaque exécution du programme, par exemple une valeur issue de la fonction `time`: `srand((int)time(NULL));` Placez cette ligne au début de chaque programme dans lequel vous voulez des séquences aléatoires différentes lors d'exécutions successives. Les fonctions `rand` et `srand` sont déclarées dans le fichier `stdlib.h`

Exercice 5. *Faites un programme qui permet de calculer et d'afficher le reste de la division d'un nombre par un autre.*

Exercice 6. *Faites un programme qui calcule la moyenne exacte des notes 9,5 8,5 et 8, la moyenne arrondie à la note la plus proche et la note arrondie à 0,5 près. Pour le dernier cas cherchez un truc qui fasse l'affaire, multipliez, redivisez, tronquez au bon moment!*

Exercice 7. *Faites un programme semblable, mais ajoutez ce qu'il faut pour qu'il lise les 3 notes au clavier.*

Exercice 8. *Faire un chronomètre qui fonctionne de la façon suivante. Quand on tape la touche RETURN, le programme lit la valeur du compteur de secondes. Puis une nouvelle fois quand on retape la même touche. Affichez des informations qui indiquent à l'utilisateur comment utiliser le chronomètre.*

Note: Pour lire une valeur a l'écran, on utilise l'instruction `scanf("%d",&valeur);` *Pour ne lire que la touche RETURN au clavier, utiliser la fonction* `getchar()`.

Exercice 9. *On suppose que les cases d'un damier sont numérotées case par case, de 0 à 63. On demande de créer un programme qui demande un numéro de case au clavier puis qui affiche la ligne et la colonne sur lesquelles cette case se trouve. (Conseil: utilisez les opérateurs de division entière et de modulo). Faire également le programme lorsque l'on numérote les cases de 1 à 64.*

Exercice 10. *On demande de créer un programme qui met 0 dans une variable, l'affiche, lui ajoute 1, l'imprime, lui ajoute 1, etc, quatre fois de suite.*

6.6 Instruction if-else, expressions logiques

La construction **`if-else`** (*si-sinon*) est la construction logique de base du langage C qui permet d'exécuter un bloc d'instructions selon qu'une condition est vraie ou fausse. Le programme ci-dessous lit un nombre au clavier et indique si l'on a tapé un nombre négatif ou positif en refixant ce nombre à une valeur positive s'il était négatif:

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

  printf("Tapez un nombre entier positif ou negatif: ");
  scanf("%d", &i);
  if (i<0) {
    i=-i;
    printf("J'ai remis i à une valeur positive.\n");
  } else {
    printf("Vous avez tapé un nombre positif.\n");
  }
}
```

Programme 11

Si la condition figurant entre parenthèses après le mot-clé **`if`** est vraie, alors le bloc d'instructions qui se trouve immédiatement après est exécuté, sinon c'est le second bloc qui se trouve après le **`else`** qui est exécuté. Le deuxième membre est facultatif, ainsi la construction conditionnelle minimale s'écrit:

```
if (condition) {
```

```
  ...
}
```

Remarquons qu'on n'utilise pas de point-virgule après l'accolade fermante d'un bloc d'instructions. Si un bloc d'instructions se réduit à une seule instruction alors on peut omettre les accolades de délimitation:

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

  printf("Tapez un nombre entier positif ou negatif: ");
  scanf("%d", &i);
  if (i>=0)
    printf("Vous avez tapé un nombre positif.\n");
  else
    printf("Vous avez tapé un nombre negatif.\n");
}
```

Programme 12

Le tableau suivant rassemble les divers opérateurs logiques opérant sur des nombres et des variables numériques:

`a < b`	Vrai si `a` strictement inférieur à `b`
`a > b`	Vrai si `a` strictement supérieur à `b`
`a <= b`	Vrai si `a` inférieur ou égal à `b`
`a >= b`	Vrai si `a` supérieur ou égal à `b`
`a == b`	Vrai si `a` strictement égal à `b`
`a != b`	Vrai si `a` différent de `b`

Tableau 5: Opérateurs logiques numériques

Des propositions logiques telles que `a<12` ou `i>=j` peuvent être combinées entre elles au moyen de connecteurs logiques `&&` (*et*), `||` (*ou*) et `!` (*négation*) pour former des expressions logiques complexes, appelées aussi *expressions booléennes*. De telles expressions sont évaluées de gauche à droite dans l'ordre naturel de lecture, l'utilisation de parenthèses permet de mieux contrôler l'ordre d'évaluation des expressions, il ne faut donc pas hésiter à les utiliser en cas de doute sur la priorité des opérateurs employés.

Par exemple les formules suivantes:

```
!(i <= 0) || (i >= 10)
```

et

```
!((i <= 0) || (i >= 10))
```

ne sont pas équivalentes Mais la dernière est équivalente à `(i > 0) && (i < 10)`, ce que le bon sens approuve (et la loi de Morgan aussi!).

Exercice 11. *Faire un programme qui demande deux nombres au clavier et qui affiche 'divisible' si le premier est divisible par le deuxième. Conseil: pensez à l'opérateur* `%`*.*

Exercice 12. *Créer une boucle qui affiche les entiers pairs de 0 à 10 et qui indique après chaque nombre s'il est divisible par 3.*

Exercice 13. *Faire un programme qui lit deux nombres et qui teste si ces nombres sont compris dans l'intervalle [-5, +5]. Sinon on affecte le premier à -5 si la première valeur donnée est plus petite que -5 et le deuxième à +5 si la deuxième valeur est plus grande que 5. Imprimer ensuite la liste des nombres, du premier au deuxième nombre.*

Exercice 14. *Faire un programme qui lit deux nombres au clavier et qui écrit la liste des nombres partant du premier et finissant au deuxième, en montant si le deuxième nombre est plus grand que le premier et en descendant sinon.*

Variable booléenne. Toute expression retournant une valeur logique ou entière est une condition valable dans une construction `if`. C'est pourquoi il est tout à fait légal et même fréquent de stocker le résultat d'une condition dans une variable de type `int` et d'utiliser plus loin cette variable comme condition d'un `if`:

```
#include <stdio.h>
void main ()
{
  float x;
  int plusgrand;

  printf("Entrez un reel: ");
  scanf("%f",&x);
  plusgrand = (x>15.0);

  if (plusgrand)
    printf ("Plus grand\n");
  else
    printf ("Plus petit\n");
}
```

Programme 13

Le langage C considère toute valeur numérique entière non nulle comme étant une valeur logique vraie, seule la valeur numérique `0` est considérée comme étant une valeur logique fausse, de ce fait l'expression `plusgrand = (x>15.0)` dépose dans la variable `plusgrand` une valeur non nulle si `x` est plus grand que `15.0` et la valeur `0` sinon. De plus, pour les mêmes raisons, une expression telle que `if (plusgrand)` n'est en fait qu'un raccourci pour `if (plusgrand != 0)`

Une variable telle que `plusgrand` dans cet exemple où l'on stocke le résultat d'une expression logique est appelée *variable booléenne*, il ne s'agit pas là d'un véritable type en soi pour le langage C, simplement d'une utilisation particulière d'une variable entière.

6.7 Instruction switch

L'instruction `switch` est l'instruction de contrôle la plus souple du langage C. Elle permet à votre programme d'exécuter différentes instructions en fonction d'une expression qui pourra avoir plus de deux valeurs. Une instruction de contrôle comme `if` ne peut évaluer que deux valeurs d'une expression: vrai ou faux. Dans le cas où l'on souhaite exécuter une action différente selon les différentes valeurs possibles d'une variable cela oblige à utiliser une cascade de `if...else` comme l'illustre le programme 14

```
#include <stdio.h>
void main ()
{
  char operation;
  int r, a, b;

  printf("Entrez un signe d'opération: ");
  scanf("%c", &operation);
  a=273; b=158; r=0;

  if (operation=='+' || operation=='p')
    r = a + b;
  else if (operation=='-' || operation=='m')
    r = a - b;
  else if (operation=='*' || operation=='f')
    r = a * b;
  else if (operation=='/' || operation=='d')
    r = a / b;
  else
    printf("Non valable: ");

  printf("%d %c %d = %d\n", a, operation, b, r);
}
```

Programme 14

L'instruction `switch` permet de résoudre ce problème de façon plus générale:

```
#include <stdio.h>
void main ()
{
char operation;
  int r, a, b;

  printf("Entrez un signe d'opération: ");
  scanf("%c", &operation);
  a=273; b=158; r=0;

  switch (operation) {
  case '+':
  case 'p':
    r = a + b;
    break;

  case '-':
  case 'm':
    r = a - b;
    break;

  case '*':
  case 'f':
    r = a * b;
    break;

  case '/':
  case 'd':
    r = a / b;
    break;

  default:
    printf("Non valable: ");
  }

  printf("%d %c %d = %d\n", a, operation, b, r);
}
```

Programme 15

Cette instruction fonctionne en examinant le contenu de la variable située après le mot-clé `switch`, dans le cas présent `operation`. Ce contenu est successivement comparé aux valeurs figurant après chacune des clauses `case`. Si une de ces clauses comporte une valeur identique alors les instructions figurant après cette clause sont exécutées y compris celles figurant après les clauses `case` suivantes. L'instruction `break` (*interrompre*) provoque une sortie immédiate du `switch`. de façon à ce que l'exécution se poursuive après l'accolade fermante du `switch`. `break` termine généralement chacun des blocs d'instructions correspondant à chacun des cas prévus car, sauf exception, on ne souhaite exécuter que les instructions figurant immédiatement après un cas donné.

Si `operation` ne correspond à aucun des cas prévus, ce sont les instructions figurant après le mot clé `default` qui sont exécutées, dans le cas présent l'affichage d'un message d'erreur au moyen de la fonction `printf`.

Les clauses `default` et `break` sont optionnelles.

Exercice 15. *Initialiser deux variables entières* `x` *et* `y` *à la valeur 200.* `x` *représente une coordonnée horizontale, et* `y` *une coordonnée verticale. Faire une boucle qui demande de taper un caractère* `d`, `g`, `h` *ou* `b` *pour droite, gauche, haut ou bas, et qui incrémente ou décrémente* `x` *ou* `y` *pour que ces coordonnées reflètent le déplacement voulu par le caractère. Pour le caractère '*`d`*', faire* `x = x + 5`. *Pour le caractère '*`g`*', faire* `x = x - 5`. *Pour le caractère '*`h`*', faire* `y = y + 5`. *Pour le caractère '*`b`*', faire* `y = y - 5`. *Afficher les valeurs des variables* `x` *et* `y` *à chaque itération de la boucle.*

6.8 Instructions while et do-while

Dans l'exercice 10, nous avons écrit quatre fois de suite la même chose. Cela peut être fait plus facilement grâce aux instructions dites de *boucle*. Ces instructions permettent de répéter un bloc d'instructions entre accolades[1], appelé *corps de boucle*, un certain nombre de fois, tant qu'une condition est vérifiée. Il existe plusieurs instructions de boucle, la plus simple est l'instruction **`while`** (*tant que*). Elle se présente sous la forme suivante:

```
while (condition) {
  ...
}
```

`do-while` (*faire-tant que*) est une instruction similaire se présentant sous la forme suivante:

```
do {
  ...
} while (condition);
```

Les instructions **`while`** et **`do-while`** présentent une différence subtile. Dans le cas de l'instruction **`while`**, la condition est tout d'abord examinée, si elle est vraie alors le bloc d'instructions est exécuté, après quoi la condition est de nouveau évaluée et le bloc d'instructions est de nouveau exécuté et ainsi de suite tant que la condition est vraie. Dans le cas de l'instruction **`do-while`**, par contre, le bloc d'instructions est d'abord exécuté **puis** la condition est évaluée. Si elle est vraie, alors le bloc d'instructions est de nouveau exécuté puis la condition de nouveau examinée et ainsi de suite. On s'aperçoit ainsi que si la condition est fausse lors de sa première évaluation alors le bloc d'instructions n'est jamais exécuté dans le cas de l'instruction **`while`** alors qu'il l'est au moins une fois dans le cas de l'instruction **`do-while`**. L'expérience montre que l'on se sert rarement de la fonction **`do-while`**, il ne faut pas oublier son existence toutefois.

Dans le programme 16, on veut que l'utilisateur rentre deux nombres dont le premier, `M`, doit être plus grand que -6 et le deuxième, `N`, doit être plus petit que 6. De plus on veut que `M` soit strictement plus petit que `N`. On demande donc à l'utilisateur de rentrer deux nombres et tant que ces nombres vérifient une condition contraire à ce que l'on souhaite, on demande à l'utilisateur de les rentrer à nouveau. On affiche ensuite les nombres compris dans l'intervalle [`M`,`N`].

1. En fait si le bloc ne comporte qu'une seule instruction, les accolades peuvent être omises. En pratique il est souvent prudent de les mettre même pour une seule instruction, car cela peut éviter quelques erreurs sournoises.

```
#include <stdio.h>
void main ()
{
  int i, M, N;

  do {
    printf("Donnez M N: ");
    scanf("%d %d", &M, &N);
  } while (M <= -6 || N >= 6 || M >= N);

  i = M;
  while (i <= N) {
    printf("%d ", i);
    i = i+1;
  }
  printf("\n");
}
```

Programme 16

Les instructions placées dans le bloc entre **`do-while`** demandent à l'utilisateur de rentrer deux nombres puis on teste les conditions d'exclusion (`M` plus petit ou égal à 6 ou bien `N` plus grand ou égal à 6 ou bien `M` plus grand ou égal `N`). Si une de ces conditions est vérifiée, on redemande les nombres à nouveau. Si ces conditions d'exclusion ne sont pas vérifiées, alors on continue plus loin.

La boucle **`while`** simple suivante a un fonctionnement similaire, à la nuance près évoquée plus haut. Avant toute chose, la condition figurant entre parenthèses est évaluée. Si elle est vraie, c'est-à-dire si la valeur courante de la variable `i` est inférieure à la valeur courante de la variable `N`, alors les instructions se trouvant entre accolades sont exécutées: la valeur de `i` est affichée et la valeur courante de `i` est augmentée de 1. La condition est alors de nouveau évaluée. Si elle est toujours vraie on exécute à nouveau les instructions du bloc. Le programme se poursuit ainsi jusqu'à ce que la condition ne soit plus vérifiée ce qui ne manquera pas d'arriver car la valeur de i est augmentée d'une unité chaque fois que le bloc d'instructions entre accolades est exécuté alors que la valeur de `N`, elle, n'est pas modifiée. Il est ainsi indispensable que la condition déterminant la fin d'une boucle comporte au moins une variable modifiée à l'intérieur de la boucle sinon la boucle s'exécute indéfiniment ou pas du tout ce qui, la plupart du temps, a des conséquences fâcheuses! Ainsi `while(0==0)` est une boucle infinie.

Exercice 16. *Ecrire un programme qui demande à l'utilisateur de deviner un nombre caché dans le programme. Le programme doit exécuter une boucle qui demande un nombre à l'utilisateur, écrire "trop grand" si le nombre tapé est plus grand que le nombre choisi, "trop petit" si le nombre est plus petit et se terminer en écrivant "vous avez trouvé" lorsque le chiffre correspond exactement à celui qu'on cherche.*

6.9 Instruction for

Souvent on souhaite exécuter un bloc d'instructions un certain nombre de fois connu à l'avance en suivant le processus habituel de comptage. Le langage C offre pour cela une instruction pratique: l'instruction **`for`**. Dans son utilisation la plus courante, elle se présente comme suit:

```
for (i=m1; i<=m2; i++) {
  ...
}
```

Cette construction permet de répéter le bloc d'instructions entre accolades, appelée *corps de boucle*, un certain nombre de fois, la variable i, appelée *compteur de boucle*, prenant une valeur différente à chaque tour de boucle: `m1` au premier tour, `m1+1` au deuxième, `m1+2` au troisième et ainsi de suite jusqu'à `m2` compris. Le corps de boucle est ainsi exécuté (`m2-m1+1`) fois. La construction i++ incrémente la valeur de la variable i de une unité (voir section 6.16).

Ainsi le programme 17 lit deux nombres au clavier et écrit les uns sous les autres une liste de nombres qui va du premier nombre tapé, au deuxième nombre tapé. On note que comme le corps de la boucle **`for`** n'est composé que d'une seule instruction alors les accolades peuvent être omises.

```
#include <stdio.h>
void main ()
{
  int i, m1, m2;

  printf("Donnez un minimum et un maximum: ");
  scanf("%d %d", &m1, &m2);
  for (i=m1; i<=m2; i++)
    printf("%d\n", i);
}
```

Programme 17

Le programme ci-dessous tabule les valeurs des sinus des 10 premiers degrés d'angle.

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

#define pi 3.14156

void main ()
{
  int i;
  float x;

  for (i=1; i<=10; i++) {
    x = sin(i*pi/180.0);
    printf("sin(%2d)  =  %f\n", i, x);
  }
}
```

Programme 18

En fait l'instruction **`for`** est plus générale que ce qui a été présenté dans les programmes 17 et 18. En effet, elle est définie plus précisément sous la forme:

```
for (initialisation; condition; continuation) {
  ...
}
```

qui n'est en fait qu'un raccourci pour la construction suivante[1]:

```
initialisation;
while (condition) {
  ....
  continuation;
}
```

`initialisation`, *`condition`* et *`continuation`* sont trois expressions pouvant être assez quelconques. Comme le montre la construction équivalente utilisant l'instruction **`while`**, *`initialisation`* est tout d'abord exécutée une seule fois. Ensuite la *`condition`* est évaluée à son tour. Si elle est vraie, alors le corps de boucle est exécuté. A la fin de l'exécution du corps de boucle, l'expression *`continuation`* est évaluée elle aussi. La *`condition`* est alors de nouveau évaluée et le corps de boucle exécuté de nouveau si elle est vraie et ainsi de suite. Dans le cas d'une boucle de comptage simple telle que nous l'avons présentée au début de paragraphe, l'expression d'*`initialisation`* consiste à initialiser une variable entière (le compteur) avec sa valeur de départ, l'expression de *`condition`* consiste à tester si la valeur du compteur est toujours inférieure à la valeur maximale souhaitée, enfin l'expression de *`continuation`* consiste à augmenter la valeur du compteur d'une unité. Comme ces expressions sont tout à fait générales, il est possible de réaliser au moyen de la boucle **`for`** toutes sortes de compteurs, par exemple la construction suivante est une boucle décroissante exécutée 5 fois, le compteur, `i`, variant de 5 à 1:

```
for (i=5; i>0; i--) {
  ...
}
```

1. Sauf du point de vue de l'instruction `continue`, voir section 6.14

La boucle suivante affiche les 5 premières puissances de 2:

```
for (i=1; i<=32; i=i*2) {
  printf(“%d ”, i);
}
```

Exercice 17. *Afficher les 10 premières valeurs d’une liste de nombres. Le premier est 0 et la liste est créée en ajoutant 1, puis 2, puis 3 etc, pour passer d’une valeur à la suivante.*

Exercice 18. *Lire deux nombres au clavier et afficher tous les nombres depuis le deuxième tapé jusqu’au premier dans l’ordre décroissant.*

6.10 Imbrication des instructions

```
#include <stdio.h>
#include <string.h>
void main ()
{
  char phrase[64];
  int i, len, diff;

  printf("Entrez une phrase: ");
  fgets(phrase,64,stdin);
  len = strlen(phrase);
  diff = ’a’ - ’A’;
  if (len >= 32) {
    for (i = 0; i <= len; i++) {
      if (’a’ <= phrase[i] && phrase[i] <= ’z’)
        phrase[i] = phrase[i] - diff;

      else if (’A’ <= phrase[i] && phrase[i] <= ’Z’)
        phrase[i] = phrase[i] + diff;
    }
  }
  printf("%s\n", phrase);
}
```

Programme 19

Toutes les instructions de C peuvent s’imbriquer les unes dans les autres. Le programme 19 transforme, dans toutes les phrases de plus de 32 caractères, les majuscules en minuscules et vice versa. Dans ce programme, l’instruction exécutée si la condition (`len >= 32`) est vérifiée est une boucle **`for`**. Les instructions répétées à l’intérieur de cette boucle sont deux tests (caractère minuscule ou caractère majuscule). A l’intérieur de chacun des deux tests se trouve une instruction d’affectation. Chaque rectangle représente l’ensemble des instructions exécutées sous le contrôle d’un test ou d’une boucle. Comme le montre ce programme, les rectangles peuvent être imbriqués. Dans l’instruction **`if`** on peut mettre d’autres instructions **`if`**, des boucles **`for`**, des affectations. A l’intérieur d’une boucle **`for`** on peut mettre d’autres boucles **`for`**, des tests, des affectations, etc...

Pour rendre compte de cette imbrication des structures et améliorer la lisibilité des programmes il est d'usage d'indenter les lignes (c'est-à-dire de les décaler d'un certain nombre d'espaces vers la droite) conformément à leur niveau d'imbrication.

6.11 Tableaux de variables

Il est souvent nécessaire de mémoriser un grand nombre d’éléments identiques. Il serait par exemple fastidieux de déclarer 1000 variables réelles représentant les 1000 valeurs qu’un signal a pris au cours du temps en vue de le traiter. De plus on ne pourrait pas faire une boucle qui utilise l’une après l’autre toutes ces variables. Pour résoudre ces problèmes, on peut utiliser des tableaux.

Un tableau se déclare de la façon suivante: `float x[200];` La variable `x` représente dans ce cas une suite de 200 variables réelles distinctes. Dans chacun des 200 éléments du tableau `x` on peut donc déposer une valeur et l’utiliser comme n’importe quelle autre variable: `x[20] = 15.0; x[30] = x[5] + 20.0;`

La valeur entière figurant entre crochets pour faire référence à un élément particulier du tableau est appelé *indice*. Le premier élément d'un tableau correspond toujours à l'indice 0. Dans une expression, l'indice peut être donné sous forme d'une constante comme ci-dessus mais peut également être une variable de type **`int`** ce qui permet, par exemple, d'imprimer la liste des variables au moyen d'une boucle **`for`**:

```
for (i=0; i<200; i++)
    printf("%f ", x[i]);
```

Si l'on veut les imprimer à 10 par ligne, il faut utiliser une double boucle du type suivant:

```
for (i=0; i<20; i++) {
  for (j=0; j<10; j++) {
    printf("%f ", x[i * 10 + j]);
  }
  printf("\n");
}
```

Vérifiez que les indices générés valent successivement 0, 1, 2, 3...

Le programme suivant présente une autre façon de faire qui évite de calculer l'expression `i*10+j` à chaque itération et qui permet par la même occasion d'afficher autant de nombres que l'on veut (pas nécessairement un multiple de 10):

```
for (i=0; i<200; i++) {
  printf("%f ", x[i]);
  if (i%10 == 0)
    printf("\n");
}
```

Il est possible de créer un tableau d'entiers, de réels, de caractères ou de n'ímporte quel autre type défini qu'il soit de base ou complexe. Il est également possible d'initialiser le contenu d'un tableau au moment de sa déclaration en utilisant une liste de valeurs entre accolades:

```
int tableau[4] = {2, 3, 12 45};
```

La représentation interne des tableaux par le langage C fait qu'il n'est pas possible de copier le contenu d'un tableau dans un autre au moyen de la simple affectation '='. Ainsi le programme suivant ne compile pas, l'affectation `y=x` étant illégale:

```
int x[3] = { 1, 3, 5};
int y[3];
y=x;
```

Pour copier un tableau dans un autre il faut copier les éléments un à un au moyen d'une boucle de type `for` ou bien utiliser la fonction `memcpy`.

Exercice 19. *Initialiser un vecteur v de 10 réels, et calculer la moyenne des 10 réels. La formule permettant de calculer la moyenne est:*

$$\bar{v} = \frac{1}{n}\sum_{i=1}^{n} v[i]$$

Exercice 20. *Initialiser un vecteur de 10 éléments, et calculer la variance des 10 éléments. La formule permettant de calculer la variance est:*

$$\sigma_v = \frac{1}{n}\sum_{i=1}^{n}(v[i]-\bar{v})^2 = \frac{1}{n}\sum_{i=1}^{n}v[i]^2 - \left(\frac{1}{n}\sum_{i=1}^{n}v[i]\right)^2$$

Exercice 21. *On demande de trier des nombres déposés dans un vecteur et de les placer dans un ordre croissant dans un autre vecteur. Déposer dans un vecteur les valeurs 3, 6, 1, 9, 2, 5 puis faire un programme qui copie dans un autre vecteur ces mêmes valeurs, mais placées dans l'ordre croissant de 1 à 9.*

Pour cela, on va faire un programme de tri simple qui effectue la boucle suivante. Chercher le minimum du vecteur et mémoriser l'indice de l'élément qui contient le minimum. Le déposer dans le vecteur des valeurs triées et remplacer ce minimum par une valeur supérieure à toutes les autres, pour qu'on ne retrouve pas à chaque boucle le même minimum. Recommencer n fois.

Exercice 22. *On peut faire le tri en utilisant un seul vecteur. Lorsque vous avez trouvé le minimum du vecteur, échangez-le avec le premier élément, puis cherchez le minimum parmi les éléments 2 à n et placez-le en 2. Puis cherchez de 3 à n, etc...*

6.12 Caractères, chaînes de caractères

Il est possible de mémoriser des caractères (lettres, chiffres, ponctuation) dans des variables de type caractère, **`char`**. Le programme suivant déclare une telle variable, y met une lettre `m`, l'affiche et lit un autre caractère. La lecture d'un caractère au clavier se fait comme celle d'un entier, au moyen de la fonction `scanf`, mais en utilisant le spécificateur de format `%c`.

```
#include <stdio.h>
void main ()
{
  char carac;

  carac = 'm';
  printf("%c", carac);
  scanf("%c", &carac);
  printf("%c", carac);
}
```

Programme 20

Le programme 21 est un exemple de manipulation de caractères:

```
#include <stdio.h>
void main ()
{
char carac, carac1, carac2;
  int integer;

  printf("Tapez un caractère :");
  scanf("%c", &carac);
  carac1 = carac-1;
  carac2 = carac+1;
  printf("%c %c %d\n", carac1, carac2, carac);
  printf("Tapez un entier: ");
  scanf("%d", &integer);
  printf("%c\n", integer);
}
```

Programme 21

Ce programme lit du clavier un caractère introduit par l'utilisateur et l'affecte à la variable `carac`, de type **`char`**. Ensuite on affecte à deux variables distinctes (`carac1` et `carac2`) le décrément et l'incrément de la variable lue. Cette opération est possible puisque dans le langage C les variables caractères de type `char` sont

assimilées à des entiers dont la valeur est celle du code ASCII correspondant au caractère (voir figure 3). On affiche le tout et on lit au clavier un nombre entier. En spécifiant `%c` dans le `printf`, nous disons au compilateur de prendre la valeur de la variable de type `integer` et l'afficher en tant que caractère `char`.

Notez que tous les caractères ne sont pas des lettres de l'alphabet ou des chiffres. Certains caractères sont appelés caractères de contrôle, et apparaissent à l'écran sous le format `^@`, `^A`, `^B`, `^C`. Ils sont utilisés pour contrôler l'affichage du texte (tabulation, retour de ligne,...).

Les variables de type `char` ne peuvent recevoir qu'un caractère.

Chaînes de caractères variables. Le langage C ne dispose pas d'un type primitif spécifique pour représenter des chaînes de caractères variables. Pour cela, le langage C utilise des tableaux de caractères. Une chaîne contenue dans un tableau de caractères se termine obligatoirement par le caractère `nul`, noté `'\0'` (backslash et zéro) de valeur numérique zéro. En pratique on n'a pas souvent à manipuler ce caractère de fin de chaîne explicitement mais il faut toutefois être toujours bien conscient de son existence. En particulier comme on doit réserver un élément de tableau pour ce caractère `nul` marquant la fin de chaîne, un tableau déclaré de `n` caractères ne peut contenir qu'une chaîne de longueur `n-1` au plus. Ainsi pour stocker une chaîne de six caractères, il faut utiliser un tableau de type **`char`** de sept éléments:

```
char chaine[7];
```

Dans le programme suivant, on a déclaré une variable appelée `votre_nom` de type "tableau de 32 caractères maximum" pouvant donc contenir des chaînes de 31 caractères maximum:

```
#include <stdio.h>
void main ()
{
  char votre_nom[32];

  printf("Comment vous appelez-vous ? ");
  scanf("%31s",stdin);
  printf("Bonjour %s\n", votre_nom);
}
```

Programme 22

Dans le programme 22 on voit comment lire un nom ou une phrase au clavier dans une variable de type tableaux de caractères en utilisant la fonction `scanf` en utilisant le spécificateur de format `%s` déjà rencontré dans la fonction `printf`. Notez également la présence d'une spécification de longueur de champ (voir section 6.3) qui limite à 31 le nombre de caractères lus au clavier par la fonction, ceci afin d'éviter de dépasser la capacité de la variable `votre_nom`, limitée à 32 caractères (un caractère doit être réservé pour le zéro marquant la fin de chaîne). L'utilisation de la fonction scanf

Il est capital de prendre garde au détail suivant concernant l'utilisation de la fonction `scanf`. Quand on lit une variable de type numérique simple (par exemple `int i;`), la syntaxe est:
`scanf("%d", &i);`

Quand on lit une variable tableau de caractères (par exemple `char s[12];`), la syntaxe est:
`scanf("%11s", s);`

Notez la présence du `&` accolé à la variable dans le premier cas et son absence dans le deuxième. Ceci est une source d'erreurs graves entraînant le plantage du programme, soyez donc **extrêmement vigilants**. Les raisons de cette différence seront expliquées au chapitre 8

La fonction `scanf`, quand elle lit une chaîne de caractères tapée au clavier, s'arrête au premier espace rencontré, les caractères suivants sont considérés comme faisant partie de la chaîne suivante si la chaîne de format indique que l'on lit plusieurs chaînes ou bien sont simplement ignorés dans le cas contraire. Il n'est donc pas possible de lire une chaîne de caractères contenant des espaces en utilisant `scanf("%s",...)`. Cette particularité est souvent gênante, c'est pourquoi il existe d'autres fonctions pour lire des chaînes de caractères au clavier qui, elles ne s'arrêtent pas aux espaces. Celle que nous utiliserons le plus souvent est la fonction `fgets`:

```
int s[12];
fgets(s,12,stdin);
```

Le premier paramètre de cette fonction est la variable désignant le tableau de caractères où la chaîne tapée doit être stockée. Le paramètre suivant est la taille de ce tableau afin que la fonction ne tente pas d'y mettre plus de caractères qu'il n'est possible et le troisième et dernier paramètre sera toujours, pour l'instant `stdin` qui désigne le clavier.

Le dépassement de la longueur des tableaux de caractères (*character buffer overflow*) est la cause d'erreurs la plus fréquente dans les programmes en C et conduit le plus souvent à un plantage du programme ou une faille de sécurité. Soyez donc très vigilants à ce sujet: à aucun moment vous ne devez mettre dans un tableau de caractères de chaîne plus grande que la taille maximale pour laquelle vous avez déclaré le tableau.

Par exemple, dans le programme 22, si on avait utilisé un appel `scanf` sans spécifier la longueur de champ: `scanf(“%s”, votre_nom);` et que l'utilisateur avait entré un nom de 32 caractères ou plus, la variable `votre_nom` se serait trouvée en situation de dépassement de capacité et le programme aurait probablement planté.On peut afficher le contenu d'une chaîne contenue dans un tableau de caractères au moyen de la fonction `printf`.

Il est possible d'initialiser le contenu d'un tableau de caractères avec une chaîne au moment de sa déclaration:

```
char prenom[32]=”Olivier”;
```

Dans une telle construction on peut omettre la taille de tableau, le compilateur utilise alors comme taille celle de la chaîne d'initialisation. Ainsi

```
char prenom[]=”Olivier”;
```

déclare le tableau `prenom` comme étant de taille huit caractères (sept caractères pour la chaîne proprement dite plus un pour le zéro final).

Attention: en dehors de l'initialisation, il n'est pas permis d'écrire `prenom=”Olivier”` (voir section 6.13)

Dans le programme suivant, on montre comment lire ou modifier un caractère particulier dans une chaîne de caractères. Nous utilisons la fonction `strlen` déclarée dans le fichier `string.h`:

```
#include <stdio.h>
#include <string.h>
void main ()
{
  char un_nom[20], caractere;
  int longueur;

  printf("Donnez un nom de plus de 4 lettres: ");
  scanf(“%19s”, un_nom);
  longueur = strlen(un_nom);
  printf("La longueur du nom est %3d\n", longueur);
  caractere = un_nom[3];
  un_nom[3] = '*';
  printf("La quatrième lettre est: %c\n", caractere);
  printf("En remplacant la lettre par * on a: %s\n", un_nom);
}
```

Programme 23

Dans ce programme on a déclaré une variable `un_nom` de type `char [20]` et une variable `caractere` de type **`char`** qui permet de mémoriser un seul caractère. Pour désigner la quatrième lettre (qui pourrait se trouver être une espace), on utilise: `un_nom[3]` (le premier élément se trouve à la position 0!).

La variable `caractere`, la variable `un_nom[3]` et la constante `'*'` sont de même type, **`char`**. On peut donc copier le contenu de l’une dans l’autre. Cela a été fait pour remplacer la quatrième lettre par une étoile dans le programme ci-dessus. La fonction `strlen(un_nom)` renvoie la longueur de la chaîne effectivement contenue dans le tableau de caractères (au plus égale à la taille du tableau moins un si aucun dépassement de capacité n'a eu lieu). Notez qu’une chaîne de longueur 1 est différente d’un char.

Exercice 23. *Lire une phrase dans une variable et compter combien il y a de mots (compter les espaces). Utiliser une boucle for, de 1 jusqu'à la fin de la phrase (*`strlen`*). Si l'on trouve la constante espace, ' ', on incrémente la variable qui compte les espaces.*

Exercice 24. *Lisez deux chaînes de même longueur et comparez-les caractère par caractère. Indiquez dans une variable booléenne si les chaînes sont égales. Conseil, utilisez une boucle for. Essayez les trois cas suivants:*

```
arbre   arbre
arbre   barbe
prix    pris
```

Ecran 5

6.13 Manipulation de chaînes de caractères

Le fichier de déclarations `string.h`, contient un certain nombre de fonctions opérant sur des chaînes de caractères comme la fonction `strlen` vue au paragraphe précédent. Nous allons ici en présenter quelques unes.

strcpy. La fonction `strcpy(dest,src)` permet de copier la chaîne de caractères `src` dans le tableau de caractères `dest`.

Le tableau de caractères de `dest` doit avoir une taille suffisante pour pouvoir contenir la chaîne `src` sinon il se produit un dépassement de capacité tel qu'il a été décrit au paragraphe 6.12.

Comme indiqué à la fin du paragraphe 6.11 il n'est pas possible de copier un tableau dans un autre au moyen de la simple affectation utilisant le signe '='. Ceci est particulièrement vrai pour les tableaux de caractères. Pour copier une chaîne dans une autre il est donc indispensable d'utiliser la fonction `strcpy(dest,src)`:

```
#include <stdio.h>
#include <string.h>
void main ()
{
  char chaine1[32], chaine2[32];
  printf("Tapez un mot: ");
  scanf("%31s",chaine1);
  chaine2 = chaine1;
  printf("Vous avez tape: %s\n",chaine2
}
```

```
#include <stdio.h>
#include <string.h>
void main ()
{
  char chaine1[32], chaine2[32];
  printf("Tapez un mot: ");
  scanf("%31s",chaine1);
  strcpy(chaine2, chaine1);
  printf("Vous avez tape: %s\n",chaine2
}
```

Programme 24

Ceci vaut également pour les chaînes constantes. Ainsi, l'affectation suivante est incorrecte:

```
char chaine[7];
chaine = "Coucou";
```

Il aurait fallut écrire:

```
strcpy(chaine,"Coucou");
```

strcmp. Des chaînes peuvent être comparées entre elles au moyen de la fonction `strcmp`. L'ordre correpond à l'ordre lexicographique (celui du dictionnaire), mais les majuscules sont séparées des minuscules. C définit l'ordre suivant: espace `0 < ... < 9 < A < ... < Z < a < ... < z` qui correspond à l'ordre donné par les valeurs ASCII des caractères. La fonction `strcmp(s1,s2)` retourne un entier positif si `s1` est alphabétiquement supérieure à `s2`, `0` si `s1` et s2 sont identiques et un entier négatif si `s1` est inférieure à `s2`.

Exercice 25. *Faire un programme qui lit des noms du clavier et qui teste à partir du deuxième, s'ils sont dans l'ordre alphabétique. Pour cela, utilisez un tableau de chaînes de caractères, que vous pouvez déclarer de la façon suivante:*

```
#include <stdio.h>
typedef char StringT [128];
void main ()
{
  StringT noms[10];
  // noms[0],noms[1],... sont des chaînes de charactères;
  // noms[3][4] est le 5ème charactère du 4ème nom
}
```

Exercice 26. *On peut faire le programme de l'exercice 25 en utilisant seulement deux chaînes de caractères. Pour cela, au fur et à mesure qu'on lit les noms au clavier, on compare le nom qu'on vient de lire (le nom courant) avec le nom précédent. Si le nom courant est plus grand que le nom précédent, le nom courant devient le nom précédent, et l'on lit un nouveau nom au clavier. Sinon le programme s'interrompt.*

Exercice 27. *Même programme que le précédent, mais avant d'utiliser l'instruction de comparaison, s'assurer que chaque caractère est majuscule. Si l'on découvre une minuscule, c'est-à-dire comprise entre 'a' et 'z', on lui soustrait la valeur 'a' - 'A'. C'est logique, puisqu'elles sont dans l'ordre, la distance entre une minuscule et sa majuscule est toujours la même.*

Attention, les caractères peuvent s'additionner. On a par exemple:

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

  dist = 'a' - 'A';
  printf("%c\n", 'N'+dist);
}
```

Programme 25

strcat. La fonction `strcat` permet de concaténer deux chaînes de caractères. Ainsi l'appel `strcat(dest,src)` permet d'ajouter la chaîne `src` à la fin de la chaîne `dest`:

```
#include <stdio.h>
#include <string.h>
void main ()
{
  char str[32];
  int length;

  strcpy(str, "abcd");
  length = strlen(str);
  printf("%d : %s\n", length, str);
  strcat (str, "efgh");
  length = strlen(str);
  printf("%d : %s\n", length, str);
}
```

Programme 26

Le tableau de caractères de destination doit avoir une taille suffisante pour pouvoir contenir la chaîne de départ plus la chaîne ajoutée sinon il se produit un dépassement de capacité tel qu'il a été décrit au paragraphe 6.12

Exercice 28. *Faire un programme qui lit un mot au clavier et qui affiche ses caractères dans l'ordre inverse, en commençant par le dernier.*

Exercice 29. *Lire une chaîne au clavier (ou déposer une chaîne constante dans une variable) et l'imprimer sans qu'apparaissent les lettres 'a' qui seraient contenues dans cette chaîne.*

Exercice 30. *Lire un nom au clavier puis l'afficher en mettant un signe - entre chaque lettre.*

Exercice 31. *Lire un mot au clavier. Afficher ensuite combien il y a de 'a' dans le mot.*

Exercice 32. *Lire une chaîne au clavier (ou déposer une chaîne constante dans une variable), composer une nouvelle chaîne de caractères qui contienne toutes les lettres de chaîne lue, sauf les 'a'. Imprimer la nouvelle chaîne de caractères.*

6.14 Les instructions break et continue

Nous avons déjà rencontré l'instruction `break` en relation avec l'instruction `switch` (voir section 6.7). En fait l'instruction `break` est plus générale et peut être utilisée également dans le corps des boucles `while`, `do-while` et `for`. Elle a pour effet dans ce cas de sortir immédiatement de la boucle et de poursuivre l'exécution du programme à l'instruction se trouvant immédiatement après l'accolade marquant la fin du corps de boucle.

Par exemple le programme suivant calcule le nombre de lettres du premier mot d'une phrase entrée par l'utilisateur:

```
#include <stdio.h>
#include <string.h>
void main ()
{
  char phrase[128];
  int i;

  printf("Entrez une phrase: ");
  fgets(phrase,128,stdin);
  for (i=0; i<strlen(phrase); i++) {
    if (phrase[i] == ' ')
      break;
  }
  printf("La longueur du premier mot est: %d\n", i);
}
```

Programme 27

La boucle `for` parcourt normalement tous les indices de `0` à `strlen(phrase)-1`, toutefois à l'intérieur du corps de boucle on teste si la lettre située à l'indice courant est une espace et si c'est le cas on interrompt immédiatement le déroulement de la boucle grâce à `break`, ainsi `i` contient la taille du premier mot.

L'instruction `continue` est similaire à l'instruction `break` mais son usage est moins fréquent. Elle sert à relancer la boucle immédiatement à l'itération suivante en ignorant les instructions qui restent encore dans le corps de boucle. Dans le cas des boucles `while` et `do-while`, la condition est immédiatement réévaluée, dans le cas de la boucle `for`, l'instruction de continuation (incrémentation du compteur en général) est exécutée immédiatement et la condition est réévaluée juste après. Le programme suivant affiche une phrase entrée par

l'utilisateur en supprimant tous les espaces et affiche ensuite le nombre de lettres (autres que des espaces) contenues dans cette phrase:

```
#include <stdio.h>
#include <string.h>
void main ()
{
  char phrase[128];
  int i, nblettres;

  printf("Entrez une phrase: ");
  fgets(phrase,128,stdin);
  nblettres = 0;
  for (i=0; i<strlen(phrase); i++) {
    if (phrase[i] == ' ')
      continue;
    nblettres++;
    printf("%c",phrase[i]);
  }
  printf("\nIl y avait %d lettres dans la phrase\n", nblettres);
}
```

Programme 28

La boucle se déroule de `0` à `strlen(phrase)-1`. A l'intérieur du corps de boucle on teste si la lettre courante est une espace, si c'est le cas l'instruction `continue` passe immédiatement à l'itération suivante en ignorant le reste du corps de boucle sinon on incrémente la variable `nblettres` et on affiche la lettre courante.

Dans de nombreux cas une utilisation judicieuse des variables booléennes permet d'éviter d'avoir recours à l'instruction `break`. Dans le programme ci-dessous, on désire déterminer si dans une chaîne de caractères déposée dans une variable, le caractère `'a'` apparaît ou non:

```
#include <stdio.h>
#include <string.h>
void main ()
{
  char un_mot[32];
  int i;

  printf("Donnez un mot: ");
  scanf("%31s", un_mot);
  for (i=0; i < strlen(un_mot); i++) {
    if (un_mot[i]=='a') {
      printf("Il y a au moins un 'a'\n");
      break;
    }
  }
  if (i == strlen(un_mot) + 1)
    printf("Il n'y a pas de 'a'\n");
}
```

Programme 29

On utilise la variable `i` pour parcourir les lettres l'une après l'autre jusqu'à ce qu'on découvre un `'a'`. La variable `i` est initialisée à 0 puis, tant qu'on n'a pas trouvé de `'a'` `i` est simplement incrémenté. On cesse d'exécuter cette boucle lorsque la variable `i` est plus grande ou égale que la longueur de `un_mot`. Dès que l'on trouve une lettre `'a'`, on provoque une sortie de boucle immédiate. Après la boucle, si l'on n'a pas trouvé de `'a'`, la variable `i` a la valeur `strlen(un_mot)+1`, sinon `i` est strictement inférieur puisqu'on est sorti de la boucle de façon anticipée. Ceci nous permet d'écrire "Il n'y a pas de a" dans le cas où il n'y en a pas. Encadrez les groupes d'instructions (boîtes) pour bien comprendre le programme.

Dans le programme précédent, on a dû se fier à la valeur de `i` pour savoir après la boucle s'il y avait un `'a'` ou non dans le mot. C'est une façon de faire qui n'est pas très explicite, elle rend donc le programme

plus difficile à comprendre. Le programme suivant a exactement le même rôle que le programme précédent, mais le fait qu'un `'a'` soit présent ou non est mémorisé dans la variable booléenne `present`:

```
#include <stdio.h>
#include <string.h>
void main ()
{
  char un_mot[32];
  int present, i;

  printf("Donnez un mot: ");
  scanf("%31s", un_mot);
  for (i=0; i <= strlen(un_mot); i++) {
    present = (un_mot[i] == 'a');
    if (present)
      break;
  }
  if (present)
    printf("Il y a au moins un 'a'\n");
  else
    printf("Il n'y a pas de 'a'\n");
}
```

Programme 30

A chaque itération de la boucle, `present` reçoit la valeur de la condition testant si la lettre examinée dans `un_mot` est un `'a'` ou pas, c'est-à-dire une valeur non-nulle si cette lettre est un `'a'` ou zéro sinon. Si `present` devient vrai alors la boucle est interrompue au moyen de l'instruction `break`.

Notez qu'un programmeur C expérimenté abrégerait le corps de boucle comme suit:

```
if (present=(un_mot[i]=='a')) break;
```

En effet une instruction d'affectation a comme valeur la valeur affectée, donc une expression telle que `present=(un_mot[i] == 'a');` a pour valeur la valeur de la variable `present` après l'affectation et peut donc être utilisée directement comme condition dans un **`if`**.

Exercice 33. *Ajoutez un test dans la condition du for qui teste si* `present` *est encore faux (négation* `present`*) pour continuer la boucle. Vous pouvez alors supprimer du corps de boucle le test sur* `present` *qui arrête la boucle au moyen du break. N'oubliez pas d'initialiser* `present` *avant la boucle faute de quoi elle pourrait ne jamais être exécutée.*

Exercice 34. *Reprenez le programme 16 qui demande à l'utilisateur de deviner un nombre. Modifier les instructions de test de façon à ce que si le nombre donné est trop grand on écrive "trop grand", s'il est trop petit on écrive "trop petit" et que sinon on mette une variable booléenne* `est_trouve` *à vrai. Cette variable sera placée également comme condition du do-while. De cette façon, la boucle ne comprendra au total que deux tests au lieu des deux if et du test du do-while. Dans le cas présent, cela ne joue pas de rôle, mais il est possible d'imaginer des programmes où le test permettant de savoir si l'on a trouvé la valeur prend beaucoup plus de temps.*

Exercice 35. *Modifier l'exercice précédent en remplaçant la boucle do-while par une boucle while.*

Exercice 36. *Créer un programme qui demande à l'utilisateur de deviner un mot. Pour cela il faut définir deux variables: une qui contient le mot à deviner et une autre qui contient une étoile à la place de chaque lettre (remplacer les lettres par des étoiles dans une boucle for). Ensuite effectuer une boucle dans laquelle on demande une lettre à l'utilisateur, puis on recherche où se trouve la lettre dans le premier mot et on remplace l'étoile située à la même position dans le deuxième mot, jusqu'à ce que les deux mots soient égaux (jeu du pendu).*

6.15 Tableaux multidimensionnels, matrices

Une matrice 2x2 de nombres réels à deux dimensions, `M`, est représentée sous la forme d'un tableau bidimensionnel que l'on déclare comme ci-dessous:

```
float M[2][2];
```

L'élément d'une matrice M_{ij} est désigné en C par `M[i][j]`. Par convention `i` désigne la ligne et `j` la colonne.

On peut initialiser un tel tableau au moment de sa déclaration grâce à la construction suivante:

```
int m[2][2] = {{3, 4},
               {0, 2} };
```

Une fois initialisé on ne peut changer les valeurs du tableau que case par case (voir section 6.11)

Exercice 37. *Déclarer deux vecteurs de dimension 2 et une matrice de dimensions 2x2. Déposer dans la matrice des valeurs qui représentent une rotation d'un angle a.*

Rappel 1: une matrice qui effectue une rotation de a radians est représentée ci-dessous:

$$\begin{bmatrix} \cos(a) & -\sin(a) \\ \sin(a) & \cos(a) \end{bmatrix}$$

Déposer dans le premier vecteur `V` *les valeurs (1.0, 0.0) et calculer dans le deuxième* `W` *le produit de la matrice* `M` *par ce vecteur. Ceci revient à calculer un deuxième vecteur déterminé par une rotation de a du premier vecteur autour du centre des axes.*

Rappel 2: le produit matriciel est donné ci-dessous:

$$W[0] = M[0][0] \cdot V[0] + M[0][1] \cdot V[1]$$
$$W[1] = M[1][0] \cdot V[0] + M[1][1] \cdot V[1]$$

Si vous faites tourner le premier vecteur d'un quart de tour (Π/2), vous devriez obtenir comme résultat (0.0,1.0). Si vous faites tourner le premier vecteur d'un huitième de tour (Π/4), vous devriez obtenir comme résultat (0.7071, 0.7071).

Exercice 38. *On ne peut faire le calcul explicite du produit matriciel que si le nombre de termes dans chaque dimension est petit. Pour de plus grandes dimensions, il faut utiliser l'instruction* `for`*, ce que nous allons introduire dans cet exercice en deux étapes.*

Première étape: remarquez qu'on peut calculer chacune des deux coordonnées du vecteur `W` au moyen d'une boucle:

```
#include <stdio.h>
void main ()
{
  float V[2] = {6.0, 5.0};
  float W[2] = {0.0, 0.0};
  float M[2][2] = {{3.2, 4.5},
                   {2.0, 5.3}};
  int i;

  for (i = 0; i < 2; i++)
    W[0] = W[0] + M[0][i] * V[i];
  for (i = 0; i < 2; i++)
    W[1] = W[1] + M[1][i] * V[i];
}
```

Programme 31

Deuxième étape: on peut faire ce même calcul au moyen de deux boucles for imbriquées dont la boucle extérieure parcourt les éléments de `W`. Cela permet de faire des produits matrice par vecteur dont la dimension est quelconque, en mettant une variable n à la place de 2.

Vous pouvez maintenant généraliser à un tableau de taille N.

Exercice 39. *On demande d'écrire un programme qui initialise une matrice M avec les valeurs M[i][j] = 3i + j, et affiche la matrice. On demande ensuite de transposer la matrice, c'est-à-dire d'échanger les éléments M[i][j] et M[j][i], pour toutes les valeurs de i et de j.*

Par exemple, si la matrice M est de taille 3, cela donnera les résultats suivants:

$$\begin{bmatrix} 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}^{T} = \begin{bmatrix} 4 & 7 & 10 \\ 5 & 8 & 11 \\ 6 & 9 & 12 \end{bmatrix}$$

Exercice 40. *On demande d'écrire un programme qui initialise une matrice M avec les valeurs M[i][j] = 3i -j, et affiche la matrice. On demande ensuite à l'utilisateur d'indiquer deux lignes de la matrice (par exemple la $1^{ère}$ et la $3^{ème}$) et d'échanger deux lignes de la matrice.*

Par exemple, échanger la ligne 1 et la ligne 3 donnera les résultats suivants:

$$\begin{bmatrix} 2 & 1 & 0 \\ 5 & 4 & 3 \\ 8 & 7 & 6 \end{bmatrix}^{EL_{13}} = \begin{bmatrix} 8 & 7 & 6 \\ 5 & 4 & 3 \\ 2 & 1 & 0 \end{bmatrix}$$

6.16 Constructions abrégées

Le C offre un certain nombre de raccourcis d'écriture pour des instructions fréquentes. Le tableau 6 en résume quelques-unes parmi les plus utilisées. Dans ce tableau `a`, `b` et `c` sont des variables numériques entières ou non:

Construction abrégée	Construction équivalente
`a++;`	`a = a+1;`
`++a;`	`a = a+1;`
`a--;`	`a = a-1;`
`--a;`	`a = a-1;`
`a += b;`	`a = a+b;`
`a -= b;`	`a = a-b;`
`a *= b;`	`a = a*b;`
`a /= b;`	`a = a/b;`

`a=(b>0) ? 12 : c;`	`if (b>0)`
	`  a = 12;`
	`else`
	`  a = c;`

Tableau 6: Constructions abrégées

Les 4 premières lignes présentent des instructions d'incrémentation et de décrémentation. On remarque que deux notations différentes existent: la notation postfixée où le signe `++` ou `--` se trouve après le nom de variable et la notation préfixée où ces signes se trouvent avant le nom de la variable.

Ces deux notations, préfixée et postfixée, incrémentent (resp. décrémentent) toutes deux la variable considérée mais présentent une différence fondamentale: l'expression `++a` incrémente `a` *avant* de prendre sa valeur alors que `a++` incrémente `a` *après* avoir pris sa valeur. Donc si `a` vaut 5, l'expression

```
b = a++;
```

met la valeur 5 dans `b` et incrémente ensuite `a` qui passe à 6 alors que l'expression

```
b = a++;
```

incrémente d'abord `a` qui passe donc à 6 et met ensuite cette valeur dans `b` qui vaut alors 6

L'opérateur ternaire `(`*`condition`*`) ? `*`val1`*` : `*`val2`* est une expression très concise qui prend la valeur *`val1`* ou *`val2`* selon que *`condition`* est vraie ou fausse.

6.17 Schémas classiques de programmation

Lorsque l'on acquiert l'expérience de la programmation, on se rend compte que les programmes que l'on écrit suivent des schémas classiques. Les trois paragraphes suivant présentent quelques schémas classiques d'utilisation des variables réelles, des chaînes de caractères, et des variables booléennes. Les trois schémas possèdent les mêmes étapes: initialisation, accumulation, affichage.

Accumulation dans une variable réelle. Beaucoup de calculs arithmétiques se résument à une somme ou un produit de plusieurs variables. Un exemple de somme est le produit scalaire, qui calcule la somme des produits des éléments de deux vecteurs a et b (tableau de réels) de dimension n, et qui peut s'écrire $x = \sum_{i=0}^{n-1} a_i b_i$. Faire une somme de plusieurs éléments revient en langage de programmation à accumuler les résultats intermédiaires dans une variable:

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

#define SIZE 4

void main ()
{
  float a[SIZE], b[SIZE], x;
  int i;

  // initialisation des vecteurs
  for (i=0; i<SIZE; i++) {
    a[i] = 100*((float)rand()/RAND_MAX);
    b[i] = 100*((float)rand()/RAND_MAX);
  }

  // affichage des vecteurs
  printf("A = ");
  for (i=0; i<SIZE; i++)
    printf("%f ", a[i]);
  printf("\nB = ");
  for (i=0; i<SIZE; i++)
    printf("%f ", b[i]);

  // accumulation du produit scalaire
  x = 0;
  for (i=0; i<SIZE; i++)
    x = x + a[i] * b[i];

  // affichage du produit scalaire
  printf("\nProduit scalaire = %f\n", x);
}
```

Programme 32

Un autre exemple d'accumulation est l'exercice 17. Dans la même catégorie, on trouve les programme qui calculent:

- la moyenne arithmétique des éléments d'un vecteur: $\bar{a} = \left(\sum_{i=0}^{n-1} a_i\right) / n$, cf. exercice 19,
- la variance des éléments d'un vecteur: $\nu = \left(\left(\sum_{i=0}^{n-1} a_i^2\right) / n\right) - (\bar{a})^2$, cf. exercice 20,
- la covariance de deux vecteurs a et b $\mu = \left(\left(\sum_{i=0}^{n-1} a_i b_i\right) / n\right) - (\bar{a}\bar{b})$,

- la factorielle d'un nombre $n! = \prod_{i=0}^{n-1} i$,
- le produit matriciel $c_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj}$,
- combinaisons $C_n^p = n!/(n-p)!p!$ ou $C_n^p = \prod_{i=1}^{p} (n-p+i) / \prod_{i=1}^{p} i$
- calcul de polynôme de degré n $y = \sum_{i=0}^{n} c_i x^i$.

Accumulation dans une chaîne de caractères. Les exercices 28 à 31 demandent simplement d'afficher un à un des caractères à l'écran. Cependant, le but d'un programme n'est pas souvent d'afficher des caractères à l'écran, mais plus souvent d'affecter une variable (c'est-à-dire lui donner une valeur), comme il l'est demandé dans l'exercice suivant. Ceci conduit à une erreur très fréquente qui est illustrée ci-dessus pour l'exercice 28:

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

void main ()
{
  char s1[32], s2[32];
  int i,len;

  printf("Chaine? ");
  fgets(s1,32,stdin);
  len = strlen(s1);
  strcpy(s2, "");

  for (i=len-1; i>=0; i--)
    s2[(len-1)-i] = s1[i];      /* accumulation */
  s2[len] = '\0';               /* Ne pas oublier le '\0' final */
  printf("\nA l'envers: %s\n", s2);
}
```

Programme 33

Dans cet exemple on construit la chaîne `s2` en ajoutant les caractères un par un dans les cases du tableau de caractères "à la main", c'est-à dire sans utiliser des fonctions de librairie comme `strcat`. Dans ce cas l'erreur consiste à oublier de marquer la fin de chaîne en omettant le caractère '\0' final (voir section 6.12). Si tel est le cas, la fonction `printf` n'est pas en mesure d'afficher correctement la chaîne `s2` à la ligne suivante et il peut même se produire un plantage du programme.

Logique booléenne. On demande d'écrire un programme qui vérifie qu'une matrice M est antisymétrique (pour tous les i et les j, M(i,j) = -M(j, i)). Cela donne généralement lieu à de nombreuses erreurs de logique:

```
// NE MARCHE PAS !!!!

#include <stdio.h>
#define SIZE 4

void main ()
{
  float m[SIZE][SIZE] =
    { { 3.0, 2.1, 4.2, 5.3},
      {-2.1, 7.4, 3.5, 6.2},
      { 4.5, 9.3,-6.4, 2.1},
      { 5.2,-3.4, 0.0, 2.3}};

  int i, j, antisym;
  antisym = 0;
  for (i=0;i<SIZE;i++) {
    for (j=0;j<SIZE;j++) {
      if (m[i][j] == -m[j][i])
        antisym ++;
    }
  }
   if (antisym)
     printf("Antisymetrique\n");
   else
     printf("Non antisymetrique\n");

  /* Ce programme teste s'il existe un
     elément m[i][j] dans la matrice tel
     m[i][j] = -m[j][i] */
}
```

```
// NE MARCHE PAS NON PLUS !!!!

#include <stdio.h>
#define SIZE 4

void main ()
{
  float m[SIZE][SIZE] =
    { { 3.0, 2.1, 4.2, 5.3},
      {-2.1, 7.4, 3.5, 6.2},
      { 4.5, 9.3,-6.4, 2.1},
      { 5.2,-3.4, 0.0, 0.0}};

  int i, j, antisym;
  for (i=0;i<SIZE;i++) {
    for (j=0;j<SIZE;j++) {
      if (m[i][j] == -m[j][i])
        antisym = 1;
      else
        antisym = 0;
    }
  }
  if (antisym)
    printf("Antisymetrique\n");
  else
    printf("Non antisymetrique\n");

  /* Ce programme ne teste que la dernière
     valeur de la matrice */
}
```

Programme 34

Les deux programmes ci-dessus ne fonctionnent pas. Par exemple, dans le programme de gauche, il suffit qu'il existe un `i` et un `j` tel que `m[i][j] == -m[j][i]` pour que la variable booléenne `antisym` devienne vraie. Le second programme indique seulement si le terme `m[SIZE-1][SIZE-1]` de la matrice est nul. La bonne façon de faire est montrée ci-dessous:

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 4
void main ()
{
  float m[SIZE][SIZE];
  int i, j, antisym;

  // Initialisation de la matrice
  for(i=0; i<SIZE; i++) {
    for(j=0; j<SIZE; j++) {
      m[i][j]=i-j;
    }
  }
```

```
  antisym = 1; //Initialisation

  // Accumulation
  for(i=0;i<SIZE;i++) {
    for(j=0;j<SIZE;j++) {
      if (m[i][j] != -m[j][i]) {
        antisym=0;
      }
    }
  }
  printf("Antisym= %d",antisym);
}
```

Programme 35

Tous les programmes qui demandent de vérifier les propriétés d'un tableau de variables ou d'une chaîne de caractères tombent dans cette catégorie: une matrice est-elle symétrique, une matrice est-elle diagonale, une matrice est-elle triangulaire supérieure, une matrice contient-elle des éléments nuls, une chaîne de caractère contient-elle des espaces, une chaîne de caractères contient-elle uniquement des lettres minuscules, uniquement des chiffres, etc...

Exercice 41. *Ecrire un programme qui vérifie qu'une chaîne de caractère est un palindrome, c'est-à-dire que la première et la dernière lettre sont identiques, la seconde et l'avant-dernière sont identiques, la troisième et l'antépénultième sont identiques, etc.*

Exercice 42. *Ecrire un programme qui vérifie que tous les éléments d'un vecteur sont ordonnés en ordre croissant.*

Exercice 43. *Ecrire un programme qui vérifie qu'une chaîne de caractère contient la lettre e.*

Exercice 44. *Ecrire un programme qui vérifie qu'une chaîne de caractère NE contient PAS la lettre e.*

Exercice 45. *Ecrire un programme qui vérifie qu'une matrice contient au moins un élément nul.*

Exercice 46. *Ecrire un programme qui vérifie qu'une matrice ne contient que des éléments positifs.*

Exercice 47. *Ecrire un programme qui vérifie qu'une matrice contient au moins un élément positif.*

Exercice 48. *Le programme 35 n'est pas des plus efficaces. Considérez une matrice m[0..999,0..999], dont l'élément m[0,0] est non nul. Après avoir observé le premier élément, on sait qu'elle n'est pas antisymétrique. En général, il est inutile de parcourir tous les éléments de la matrice dès que l'on est sûr qu'elle n'est pas antisymétrique. Pouvez-vous modifier le programme 35 de façon à ce que le programme s'interrompe dès que la variable* `antisym` *a pris la valeur logique fausse. Il y a deux façons de faire: soit modifier la condition de boucle soit utiliser l'instruction* `break`.

7 Structurer l'exécution d'un programme: les fonctions

Il est pratiquement impossible d'écrire un programme compliqué en alignant un grand nombre d'instructions C de base. On se rend bien compte que dans le cas de programmes de calculs, des opérations telles que le produit matriciel vont se répéter souvent et, dans les programmes graphiques, l'affichage de formes simples (lignes, rectangles, cercles) se répétera aussi. Il serait donc intéressant d'écrire une fois pour toutes le petit morceau de code qui décrit une opération courante, et qu'on puisse l'exécuter un peu partout dans le programme principal. C'est ce qu'offre le concept de *fonction* ou *sous-routine* (parfois simplement *routine*). Suivant les situations, on peut simplement reprendre des fonctions générales qui existent déjà, ou en écrire soi-même.

Les sections suivantes vous apprendront à utiliser des routines graphiques existantes (sections 7.1 et 7.2), des fonctions de calcul matriciel (section 7.3) et des fonctions mathématiques existantes (section 7.4), puis à écrire vous même des fonctions (section 7.5).

Le chapitre 11 contient deux sections avancées sur les fonctions, la première expliquant les modules et la compilation séparée (section 12), et la seconde décrivant des routines d'interaction avec l'écran graphique, permettant notamment de créer des jeux simples (section 13).

7.1 Module graphique

Le but de cette section est de vous apprendre deux choses: d'une part l'utilisation d'une bibliothèque de routines (*library*) d'autre part le contenu d'une bibliothèque d'affichage à l'écran. Considérons le programme 36. Après la directive habituelle `#include <stdio.h>`, il contient une autre directive

`#include`, suivie d'un nom de fichier entre guillemets, `"Graphics.h"`. Le fait d'utiliser ici des guillemets plutôt que <> indique juste que `Graphics.h` n'est pas une librairie standard.

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

void main ()
{
  FillRectangle (100, 100, 200, 300) ;
  printf("Pressez Return pour terminer le programme\n");
  getchar () ;
}
```

Programme 36

Le fichier `Graphics.h` est ce qu'on appelle un fichier d'*interface* (*header file*). Il contient une liste d'interfaces de fonction précisant le nombre et le type des paramètres ainsi que le type de retour de chaque fonction. L'intérêt des fonctions est qu'elles agissent comme des boîtes noires: il n'est pas nécessaire de savoir comment elles fonctionnent en interne pour pouvoir s'en servir, une simple description de ce qu'elles font ainsi que la connaissance de leur interface suffit.

```
void FillRectangle (int left, int top, int right, int bottom) ;
void DrawRectangle (int left, int top, int right, int bottom) ;
void FillOval (int left, int top, int right, int bottom) ;
void DrawOval (int left, int top, int right, int bottom) ;
void DrawLine (int x0, int y0, int x1, int y1) ;
void FillTriangle (int x0, int y0, int x1, int y1, int x2, int y2) ;
void DrawTriangle (int x0, int y0, int x1, int y1, int x2, int y2) ;
void PenSize (int pixels) ;
void SetColor (int color) ;
void SuspendRefresh () ;
void ResumeRefresh () ;
void SetWindowSize (int sizeX, int sizeY) ;

void Delay (int millisec) ;
```

Programme 37

Par exemple, l'interface de la fonction `FillRectangle` (première ligne du programme 37) a quatre paramètres de type entier, **`int`**, appelés `left`, `top`, `right` et `bottom`. Elle dessine dans une fenêtre graphique un rectangle plein dont le coin supérieur gauche a pour coordonnée (`left`, `top`), et le coin inférieur droit (`right`, `bottom`). Le programme 36 contient un appel à la fonction `FillRectangle`. Un appel de fonction est un nom de fonction suivi de *paramètres effectifs*. Il faut que le nombre et le type de paramètres effectifs correspondent au nombre et au type de chacun des *paramètres formels*. En l'occurrence, il faut spécifier 4 valeurs entières entre parenthèses dans l'appel de la fonction `FillRectangle`.

Une fonction est en fait un emballage pour une série d'instructions (parfois très longue) qui effectue la commande suggérée par le nom de la fonction. La série d'instructions qui correspond au nom de la fonction s'appelle le *corps de la fonction*. Dans les sections 7.1 à 7.4, nous nous contenterons de lire des interfaces de fonctions et de faire des appels de fonctions. Dans la section 7.5 nous écrirons des interfaces et corps de fonctions et les appellerons.

Pour résumer, il y a trois notions importantes liées à l'utilisation des fonctions: l'*interface* (nom de la fonction et liste de ses paramètres), le *corps* (la liste des instructions qui effectuent la commande suggérée par le nom de la fonction), et l'*appel de fonction* (l'utilisateur demande que la fonction soit utilisée avec une valeur spécifique pour chacun des paramètres).

Les autres fonctions contenues dans le fichier `Graphics.h` sont `DrawRectangle` (contour d'un rectangle); `FillOval` (ovale plein); `DrawOval` (contour d'un ovale); `DrawLine` (trait); `PenSize` (épaisseur du trait); `SetColor` (niveau de gris, de 0 (noir) à 8 (blanc)). La routine `SetWindowSize` donne la taille initiale de la fenêtre et doit être appelée avant toutes les autres.

Les routines `SuspendRefresh` et `ResumeRefresh` sont des routines d'optimisation de la performance. Lorsqu'on a beaucoup de primitives graphiques (rectangles, lignes, cercles) à dessiner, il vaut mieux d'abord interrompre le rafraîchissement d'écran (`SuspendRefresh`), dessiner toutes les primitives, et relancer le rafraîchissement d'écran qui affichera alors toutes les primitives d'un coup.

Le fichier `Graphics.h` déclare également la routine `Delay` qui permet de suspendre l'exécution d'un programme pendant un certain nombre de millisecondes. Cela est utile pour simuler le mouvement (afficher un point, attendre 50ms, l'effacer, et afficher le point à un autre endroit).

Pour compiler le programme 36, on utilise les commandes suivantes:

```
cosun12% gcc -g -o grp graphics-first.c -l$GLIB
```

Dans la ligne de commande, vous reconnaissez la ligne de commande classique

```
gcc -g -o grp graphics-first.c
```

On doit ajouter l'option `-l$GLIB` (moins PETIT ell dollar glib majuscules) à la fin de la commande[1].

Le point `(0,0)` d'une fenêtre graphique se trouve par convention dans le coin supérieur gauche. La coordonnée `x` croît vers la droite. La coordonnée `y` croît vers le bas (figure 4). La figure 4 montre aussi la fenêtre graphique résultant de l'exécution du programme 36.

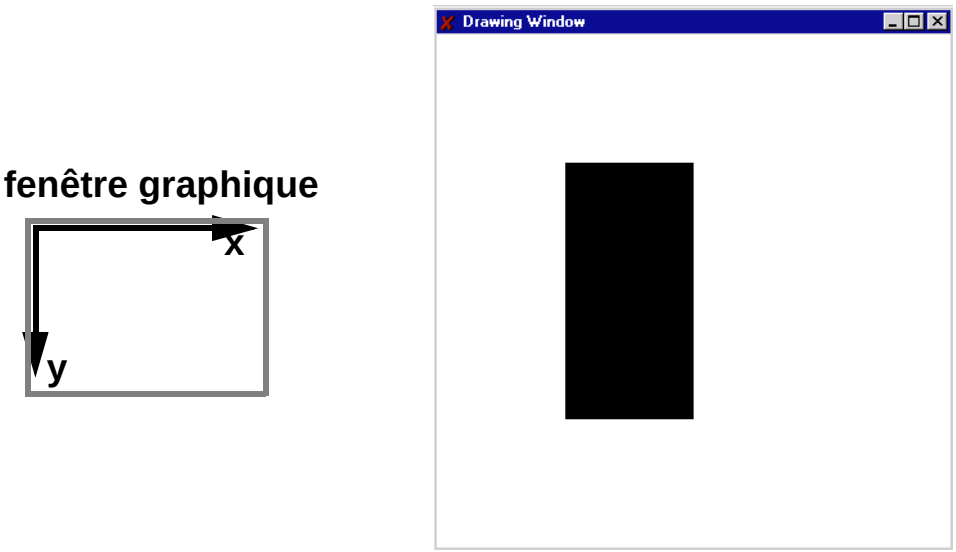
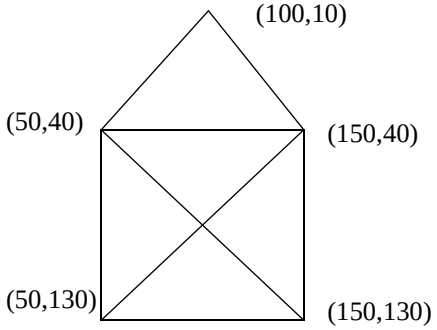


FIGURE 4: Fenêtre graphique

1. Pour les esprits curieux, `GLIB` est une variable d'environnement de la fenêtre terminal (à ne pas confondre avec les variables de vos programmes C). On peut en afficher le contenu en utilisant dans la fenêtre terminal la commande `echo $GLIB`. Cette variable d'environnement permet au compilateur C de trouver comment exécuter chacune des fonctions utilisées, externes au programme.

Exercice 49. *Faire le dessin d'une maison dont les points sont donnés ci-dessous:*



Exercice 50. *Lire une valeur au clavier et dessiner la maison de sorte que le coin en bas à gauche soit placé au même endroit et que la maison soit dessinée à l'échelle donnée par le nombre lu.*

Exercice 51. *Dans cet exercice, on demande de dessiner un sinus qui fait deux oscillations sur la largeur de l'écran (0 à 400).*

Exercice 52. *Calculer le minimum de la fonction sin(x) pour x variant entre 0 et 4.0 par pas de 0.01. Pour cela mettez 1.0 dans la variable min, puis pour chaque valeur de x, testez si la valeur sin(x) est plus petite que min. Déposez cette valeur dans min si c'est le cas.*

Exercice 53. *Dans cet exercice, on demande d'afficher la fonction sin(x) * sin(x + Pi/3) de façon que le tracé tienne entre les horizontales 20 et 180. Cherchez le maximum et le minimum de la fonction dans une première boucle puis affichez la fonction à l'échelle.*

Exercice 54. *Dessiner des figures de Lissajou sur la fenêtre graphique. Les figures de Lissajou sont des figures créées par le déplacement d'un point dont les coordonnées en x et en y varient selon des fonctions sinus de différentes fréquences. On calcule donc x = sin(t) et y = sin(*`cste`**t) pour des valeurs croissantes de t, et on affiche les points (x;y) proprement décalés pour qu'ils apparaissent dans l'écran.* `cste` *est une constante réelle dont la valeur est choisie par l'utilisateur.*

Exercice 55. *Reprendre l'exercice 38, mais afficher dans la fenêtre graphique un cercle de 5 pixels de rayon à la position donnée par les coordonnées x et y. Pourquoi le point descend-il lorsque vous appuyez 'h'.*

7.2 Module d'affichage de fonctions

On se rend compte, dans les exercices de la section 7.1, qu'ajuster les fonctions dans la fenêtre graphique présente une certaine difficulté, et surtout que cette difficulté se répète pour chaque nouvelle fonction. La difficulté provient de ce qu'il faut convertir les coordonnées de *graphe* (réels, par exemple (t, sin(t))) en coordonnées d'*écran* (entiers, par exemple (50,150)). Le module introduit dans cette section permet de contourner la difficulté.

Le fichier `Graphs.h` contient la liste des fonctions aidant à l'affichage des fonctions mathématiques. Il utilise lui-même le module `Graphics.h`. (programme 38):

```
void SetScreenSize(int l,int t,int r,int b);
void SetGraphSize(float l,float t,float r,float b);
void DrawGraphLine(float fromGX, float fromGY, float toGX, float toGY);
void DrawGraphPoint(float hereX, float hereY, int radius);
```

Programme 38

La routine `SetScreenSize` définit la partie de la fenêtre graphique utilisée pour dessiner la fonction. La routine `SetGraphSize` définit la taille du graphe que l'on souhaite afficher. Les routines `DrawGraphLine` et

`DrawGraphPoint` convertissent automatiquement les coordonnées de graphes en coordonnées d'écran, sans avoir à faire les conversions explicitement, comme à la section précédente.

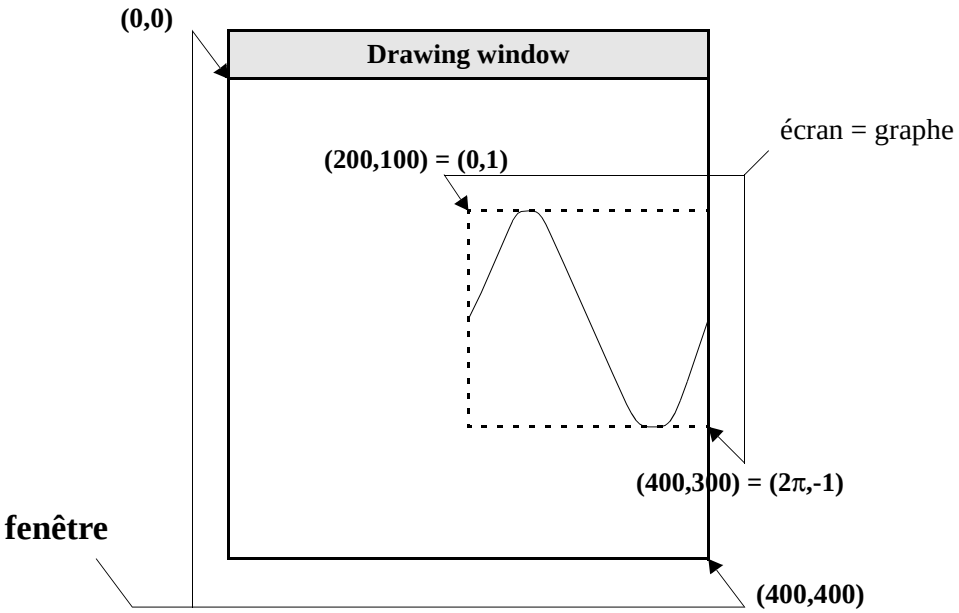


FIGURE 5: Fenêtre, écran, graphe (window, screen, graph)

La figure 5 montre, dans une fenêtre graphique de taille (400,400), un écran (c'est-à-dire une sous-fenêtre) de taille (200,200) dont le coin supérieur gauche a pour coordonnées (200,100). Dans cet écran, on dessine la fonction sinus dans l'intervalle (0,2π). La taille de la fenêtre graphique est définie à l'aide de la fonction `SetWindowSize` (module `Graphics.h`). Les tailles d'écran et de graphe (définis à l'aide des fonctions `SetScreenSize` et `SetGraphSize`) servent simplement à définir les rapports entre les coordonnées d'écran et de graphe. Dans le cas de la figure 5, le point (0,1) dans le graphe correspond à la coordonnée (200,100) dans la fenêtre graphique, et le point (2π, -1) dans le graphe correspond à la coordonnée (400,300) dans la fenêtre graphique.

Le programme 39 permet d'afficher la fonction sinus entre 0 et 2π en x, et entre -1 et 1 en y, dans la fenêtre 200,100,400,300:

```
#include <stdio.h>
#include "Graphics.h"
#include "Graphs.h"
#include "math.h"

#define PI 3.14159

void main ()
{
  float x;

  SetScreenSize(200,100,400,300);
  SetGraphSize(0,1,2*M_PI,-1);
  for (x=0; (x + 0.01) <= (2*M_PI); x = x + 0.01) {
    DrawGraphLine (x,sin(x),x+0.01,sin(x+0.01));
  }
  getchar();
}
```

Programme 39

Comme précédemment, pour compiler ce programme, vous devez utiliser la commande de compilation:

```
cosun12% gcc -g -o drawsinus drawsinus.c -l$GLIB
```

Exercice 56. *Dans le programme 39, la fonction sinus est appelée deux fois à chaque itération de la boucle while. Réécrivez ce programme de façon à n'appeler qu'une fois la fonction à chaque itération de la boucle.*

Exercice 57. *Dans cet exercice, on demande d'afficher la fonction sin(x) * sin(x + Pi/3) de façon que le tracé tienne entre les horizontales 20 et 180 (voir exercice 53). Cette fois-ci, utilisez les routines du module Graphs.h*

Exercice 58. *On demande d'afficher dans la même fenêtre graphique deux fonctions: d'une part la fonction sinus entre* -Π *et* Π*, dans la sous-fenêtre (20,120,180,280) ; et d'autre part la fonction logarithme naturel (log), entre 0.1 et 3.1, dans la sous-fenêtre (220,120,380,280).*

7.3 Module de calcul matriciel

Le fichier `matrix.h` (Prog. 40) contient les déclarations des fonctions de calcul matriciel mises à votre disposition.

```
VectorPT CreateVector(unsigned int size);
void DeleteVector(VectorPT mpt);

MatrixPT CreateMatrix(unsigned int lines, unsigned int cols);
void DeleteMatrix(MatrixPT mpt);

double GetVectorElement(VectorPT vpt, unsigned int index) ;
void SetVectorElement(VectorPT mpt, unsigned int index, double value);

double GetMatrixElement(MatrixPT mpt, unsigned int line, unsigned int col) ;
void SetMatrixElement(MatrixPT mpt, unsigned int line, unsigned int col,
                      double value);

void WriteVector (VectorPT vpt);
void WriteMatrix (MatrixPT mpt);

double ScalarProduct (VectorPT v1, VectorPT v2);
void MatVectMult (MatrixPT mpt, VectorPT vpt, VectorPT rpt );
void MatMatMult (MatrixPT mpt1, MatrixPT mpt2, MatrixPT rpt);

void SolveSystem (MatrixPT spt, VectorPT rpt);
```

Programme 40

Cet ensemble de fonctions permet de manipuler des vecteurs et des matrices $m \times n$. Deux nouveaux types sont ainsi définis `VectorPT` et `MatrixPT`. Pour se servir d'une matrice ou d'un vecteur il faut d'abord non seulement le déclarer mais aussi le "créer" au moyen des fonctions `CreateVector` et `CreateMatrix`. Quand on a fini de s'en servir il faut le détruire au moyen de `DeleteMatrix` ou `DeleteVector`. Par exemple, pour utiliser une matrice de 3 lignes et 2 colonnes:

```
MatrixPT myMatrix;
myMatrix = CreateMatrix(3,2);
...
DeleteMatrix(myMatrix);
```

Pour utiliser un vecteur à 3 composantes:

```
VectorPT myVector;
myVector = CreateVector(3);
...
DeleteVector(myVector);
```

On peut consulter et fixer les valeurs d'une matrice ou d'un vecteur au moyen des fonctions `GetVectorElement`, `SetVectorElement`, `GetMatrixElement` et `SetMatrixElement`.

Les fonctions `WriteVector` et `WriteMatrix` affichent respectivement le contenu d'un vecteur ou d'une matrice dans la fenêtre terminal. La fonction `ScalarProduct` calcule et retourne le produit scalaire de deux vecteurs de taille identique `v1` et `v2`. La fonction `MatVectMult` multiplie le vecteur `vpt` par la matrice `mpt` et

met le vecteur résultant dans le paramètre `rpt` pourvu que les tailles des vecteurs et de la matrice correspondent. **Attention**: le vecteur `rpt` doit avoir été préalablement initialisé et créé au moyen de la fonction `CreateVector`. La procédure `MatMatMult` multiplie les deux matrices `mpt1` et `mpt2`, et met la matrice résultante dans la matrice `rpt`. **Attention**: là encore la matrice résultat `rpt` doit avoir été préalablement initialisé et créée au moyen de la fonction `CreateMatrix`. La fonction `SolveSystem` résout une système linéaire A.x = b de taille n, et met le résultat dans le vecteur `rpt`. Pour un système de taille n, le paramètre `s` est une matrice de taille (n,n+1) (n lignes, n+1 colonnes), dont les n premières colonnes contiennent la matrice carrée A, et dont la dernière colonne est le vecteur b. **Attention**: de nouveau le vecteur `rpt` doit avoir été préalablement initialisé et créé à la bonne taille au moyen de la fonction `CreateVector`.

Le programme 41 montre l'utilisation de quelques routines du module de calcul matriciel. Il s'agit simplement d'un programme qui réalise l'exercice 60 (rotation d'un vecteur à l'aide du calcul matriciel) en utilisant les fonctions déclarées dans le fichier d'en-tête `matrix.h`. On peut remarquer que l'on s'est débarrassé de toutes les boucles et donc des risques d'erreurs.

```
#include <stdio.h>
#include "matrix.h"
#include "math.h"

void main ()
{
  MatrixPT m;       /* Declaration de la matrice m */
  VectorPT v, w;    /* Declaration des vecteurs v et w*/
  double a;

  a = M_PI / 4;
  m = CreateMatrix(2,2);   /* Création de la matrice m */
  v = CreateVector(2);     /* Création du vecteur v */
  w = CreateVector(2);     /* Création du vecteur w */
  SetVectorElement(v,0,1);
  SetVectorElement(v,1,0);
  SetMatrixElement(m,0,0,cos(a));
  SetMatrixElement(m,0,1,-sin(a));
  SetMatrixElement(m,1,0,sin(a));
  SetMatrixElement(m,1,1,cos(a));
  printf("m = \n"); WriteMatrix(m);
  printf("v = \n"); WriteVector(v);
  MatVectMult (m,v,w);
  printf("w = \n"); WriteVector(w);
  DeleteMatrix(m);
  DeleteVector(v);
  DeleteVector(w);
}
```

Programme 41

On remarque que le programme débute par la déclaration des variables matrice et vecteurs et qu'ensuite ils sont créés par les appels à `CreateMatrix` et `CreateVector`. Des valeurs sont ensuite fixées grâce aux fonctions `SetVectorElement` et `SetMatrixElement`.

Pour pouvoir utiliser le module de calcul matriciel, copiez les fichiers `matrix.h` dans `~/include` et `libmatrix.a` dans `~/lib` s'ils ne s'y trouvent déjà pas. Ces fichiers se trouvent dans `~gennart/include` et `~gennart/lib`. La ligne de compilation requise pour compiler un programme `testmat.c` qui utilise le module de calcul matriciel est alors:

```
cosun12% cc -g -o tesmat testmat.c -lm -lmatrix
```

Exercice 59. *Tester les routines de calcul matriciel à l'aide du programme suivant. Initialiser la matrice s d'un système A.x = b avec les valeurs A(i,j) = sin(i)*cos(j) et b(i) = i. Résoudre le système à l'aide de la routine SolveSystem, ce qui vous donne un vecteur x. Vérifier que le résultat est juste en multipliant la matrice A par le*

vecteur x, et en imprimant le résultat. Vous devriez retrouver les valeurs du vecteur b. A chaque étape de cet exercice, utilisez autant que possible la routine WriteMatrix.

Exercice 60. *On demande de tracer les points générés par la rotation du vecteur défini dans l'exercice 37. Pour cela, créer une fonction qui affiche un point dont les coordonnées sont passées en paramètre (vecteur). Dans le programme principal, initialiser une matrice de rotation M, initialiser un vecteur à (80, 0) et faire une boucle de 1 à 100 appelant la procédure de multiplication de matrice puis la procédure qui dessine le vecteur. A chaque boucle, transférer le résultat dans le vecteur placé en deuxième position dans MatVectMult.*

Exercice 61. *Faire une horloge dont les aiguilles sont représentées par des points de grandeurs différentes. Pour cela utilisez si nécessaire les trois modules introduits jusqu'ici.*

7.4 Le module de fonctions mathématiques

Le dernier module que nous allons utiliser dans le cadre de ce cours contient des fonctions mathématiques. Certaines des fonctions mathématiques présentées ici ont déjà été introduites dans la section 6.5

```
#defineM_E2.7182818284590452354
#defineM_LOG2E1.4426950408889634074
#defineM_LOG10E0.43429448190325182765
#defineM_LN20.69314718055994530942
#defineM_LN102.30258509299404568402
#defineM_PI3.14159265358979323846
#defineM_PI_21.57079632679489661923
#defineM_PI_40.78539816339744830962
#defineM_1_PI0.31830988618379067154
#defineM_2_PI0.63661977236758134308
#defineM_2_SQRTPI1.12837916709551257390
#defineM_SQRT21.41421356237309504880
#defineM_SQRT1_20.70710678118654752440

extern double acos (double);
extern double asin (double);
extern double atan (double);
extern double atan2 (double, double);
extern double cos (double);
extern double sin (double);
extern double tan (double);

extern double cosh (double);
extern double sinh (double);
extern double tanh (double);

extern double exp (double);
extern double frexp (double, int *);
extern double ldexp (double, int);
extern double log (double);
extern double log10 (double);
extern double modf (double, double *);

extern double pow (double, double);
extern double sqrt (double);

extern double ceil (double);
extern double fabs (double);
extern double floor (double);
extern double fmod (double, double);
```

Programme 42

Une fonction peut être utilisée dans une expression quelconque, comme l'illustre le programme 43, qui montre dans un cas particulier que la tan(θ) est effectivement égale à sin(θ)/cos(θ). On remarque qu'en langage C, on peut appeler une fonction à l'intérieur même d'une autre fonction. Ici, il n'est pas nécessaire de faire le calcul puis de le mettre dans une variable avant d'afficher la variable. On peut simplement mettre le calcul à effectuer

à l'endroit où on met la valeur à afficher. Ceci est dû au fait que les fonctions mathématiques retournent une valeur (voir section 7.5).

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

void main ()
{
  float theta = 1.0;
  printf("tan(theta) = %8.3f %8.3f\n", tan(theta), sin(theta)/cos(theta));
}
```

Programme 43

La librairie mathématique du langage C est normalisée, pour l'utiliser il faut simplement rajouter l'option `-lm` lors de la compilation:

```
cosun12% gcc -g -o program43 program43.c -lm
```

7.5 Fonctions

Jusqu'à présent, nous nous sommes contentés d'utiliser des fonctions existantes. On se rend compte, au fur et à mesure que l'on écrit des programmes plus complexes, que l'on réécrit souvent les mêmes instructions C pour effectuer la même tâche. C'est alors le moment de commencer à écrire ses propres fonctions.

Une fonction est un morceau de programme auquel est attribué un nom. On peut appeler cette fonction depuis un point quelconque du programme principal ou d'une autre fonction en mentionnant simplement son nom. On distingue donc d'une part la définition de la fonction et d'autre part son appel.

Pour illustrer notre propos, nous allons écrire une fonction qui convertit des degrés fahrenheit en degrés celsius (Prog. 44):

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

int FahrToCelsius(int fahr);

int tempc;

void main ()
{
  int tempf = 35;

  tempc = FahrToCelsius(tempf);
  printf("%d F -> %d C\n",tempf,tempc);
  printf("%d F -> %d C\n", 12, FahrToCelsius(12));
}

int FahrToCelsius(int fahr)
{
  return 5*(fahr-32)/9;
}
```

déclaration de la fonction

appels de la fonction

définition de la fonction

Programme 44

On remarque que la fonction `FahrToCelsius` a la même structure que le programme principal définit par `main`. Cela n'a rien d'étonnant car `main` est une fonction à part entière qui n'a qu'une seule particularité: celle de s'exécuter automatiquement au lancement du programme alors que les autres fonctions doivent être appelées explicitement depuis une autre fonction pour s'exécuter.

Déclaration et définition. Une *définition* de fonction commence par un mot-clé de type qui correspond au type de la valeur de retour, le type particulier **`void`** indiquant que la fonction ne retourne rien. Viennent ensuite le nom de la fonction suivi de paramètres placés entre parenthèses. Ce sont les *paramètres formels* car ils définissent la forme que va prendre l'appel de la fonction, c'est-à-dire le nombre et le type de ces paramètres. On

peut avoir de 0 à n paramètres formels. Chaque paramètre formel est défini par le type de la variable suivi du nom de la variable. Cet ensemble: nom de la fonction, type de retour, nombre et type des paramètres s'appelle l'*interface*, l'*en-tête*, le *prototype* ou la *signature* de la fonction. La partie suivante de la définition d'une fonction, placée entre une accolade ouvrante et une accolade fermante, constitue le *corps de fonction*. Sa structure est la même que celle du corps de la fonction `main`: tout d'abord une suite optionnelle de déclarations de variables propres à la fonction, suivie d'instructions à exécuter. Parmi ces instructions, l'instruction **`return`** a un effet particulier: elle interrompt immédiatement le déroulement de la fonction en renvoyant la valeur de l'expression située à sa droite qui se trouve ainsi être la valeur de retour de la fonction. Cette expression doit donner une valeur du type déclaré comme type de retour par l'interface de la fonction.

Appel de fonction. Lors d'un appel de fonction le compilateur vérifie le nombre et le type de paramètres qui lui sont passés et s'assure que la valeur de retour de la fonction est correctement employée. Si la fonction renvoie une valeur, l'appel doit être placé là où une variable du type retourné est autorisée. Pour que ces vérifications puissent avoir lieu, toute fonction doit être *déclarée* ou *définie* **avant** d'être appelée. C'est pourquoi il est courant de définir l'ensemble des fonctions avant la fonction `main`. Si une fonction n'est pas définie avant l'endroit où elle est appelée, alors elle **doit** au moins être déclarée. Une déclaration de fonction est constituée d'un simple rappel de son interface suivi d'un point-virgule (voir programme 44). Lorsque l'on écrit une bibliothèque de fonctions (c'est-à-dire un ensemble de fonctions relatives à un même thème), il est d'usage de regrouper les déclarations de ces fonctions dans un fichier dont l'extension est `.h`. Nous avons rencontré un certain nombre de ces fichiers jusqu'ici: `stdio.h`, `matrix.h`, `Graphics.h`. La directive `#include` insère ces fichiers de déclarations ce qui permet d'utiliser les fonctions déclarées dedans dans la suite du programme.

Dans le cas du programme 44, lorsque l'exécution parvient à l'appel `FahrToCelsius(tempf)`, le programme copie les valeurs des paramètres de l'appel, appelés *paramètres effectifs* dans ceux de la définition, appelés paramètres formels. En l'occurrence, ici, la valeur de la variable `tempf` est copiée dans la variable `fahr` pour la durée de l'exécution de la fonction. Donc pendant l'exécution du corps de cette fonction `fahr` prend la valeur 35. Lors de l'appel `FahrToCelsius(12)`, `fahr` prend la valeur 12 pendant l'exécution du corps de la fonction.

Lors d'un appel tel que `FahrToCelsius(tempf)`, c'est bien la valeur de la variable `tempf` au moment de l'appel, c'est-à-dire le nombre 32, qui est copiée dans `fahr` pour être ainsi passée à la fonction et non pas la variable `tempf` elle-même. Ainsi, si le contenu de la variable `fahr` était modifié à l'intérieur de la fonction `FahrToCelsius` cela n'aurait aucune influence sur la valeur de la variable `tempf` qui continuerait de valoir 32. C'est pourquoi on dit que le C passe les paramètres aux fonctions *par valeur*. Ainsi il n'est a priori pas possible de modifier la valeur d'une variable extérieure de l'intérieur d'une fonction. En fait cela est possible en passant comme paramètre non pas une variable elle-même mais un pointeur sur elle (voir section 8.5)

Dans l'appel de la fonction les paramètres effectifs sont séparés par des virgules et le type n'apparaît pas (car il est déjà défini par la déclaration de la fonction). Les paramètres effectifs peuvent avoir les mêmes noms que les paramètres formels. Mais en principe on appelle la même fonction avec différents jeux de paramètres effectifs, ce qui justifie d'avoir les deux sortes de paramètres.

Exercice 62. *Ecrire un programme qui définit et appelle plusieurs fois une fonction qui convertit des degrés minutes secondes en radians. La formule est:*

$$radians = \frac{\pi}{180}\left(degres + \frac{minutes}{60} + \frac{secondes}{3600}\right)$$

Exercice 63. *Ecrire une fonction qui calcule la valeur y d'un polynôme de degré 3, sachant la valeur des coefficients c_0, c_1, c_2, c_3 et l'abscisse x.*

Pour l'exercice 63, la formule du polynôme est bien entendu $y = c_0 + c_1x + c_2x^2 + c_3x^3$. Comment calculer la $n^{ième}$ puissance de x? On peut bien entendu utiliser des formules sophistiquées basées sur les logarithmes (attention aux nombres négatifs), ou trouver un module qui contient une fonction qui calcule y^x, avec y et x

réels. Cependant, dans le calcul du polynôme, on sait que l'exposant est entier. Il vaut donc mieux utiliser une boucle qui calcule x^i, en initialisant x à 1, et en multipliant i fois par x. On peut cependant faire mieux, en factorisant le polynôme de la façon suivante: $y = c_0 + x\,(c_1 + x\,(c_2 + c_3 x))$. Ceci réduit le nombre de multiplications à n pour un polynôme de degré n, contre $n*(n+1)/2$ pour la formule non factorisée. Utilisez si possible la formule factorisée pour calculer le polynôme.

Exercice 64. *Créer une fonction qui calcule le nombre de combinaisons d'un ensemble de n pièces prises m par m:*

$$\binom{m}{n} = \frac{n!}{m! \cdot (n-m)!} = \frac{\prod_{i=0}^{m-1} n-i}{\prod_{i=1}^{m} i}$$

Attention, la fonction factorielle donne des résultats qui dépassent rapidement la capacité des entiers de type `int`. *Il vaut mieux utiliser la formule simplifiée qui calcule le rapport de deux produits.*

Exercice 65. *Ecrire et tester une fonction qui calcule la norme d'un vecteur (x,y).*

Imbrication des fonctions. En langage C, on ne peut pas définir de fonction a l'intérieur d'une autre fonction. Toutes les fonctions doivent être définies au même niveau.

Variables globales et variables locales. On peut déclarer des variables à l'intérieur d'une fonction comme nous l'avons fait jusqu'ici dans la fonction `main`. De telles variables ne sont visibles que dans la fonction où elles sont déclarées, elles n'ont pas d'existence ailleurs et le compilateur signale une erreur si l'on tente de s'en servir ailleurs. On appelle ces variables, variables de fonction ou *variables locales*, par opposition aux variables déclarées en dehors de toute fonction (y compris la fonction `main`) qui, elles, sont visibles dans tout le programme après leur déclaration, c'est-à-dire dans toutes les fonctions. On appelle ces variables, variables de programme ou *variables globales.*

Le programme 45 est pratiquement identique au programme 44. Il montre l'utilisation de d'une variable locale, `celsius`, dans la fonction `FahrToCelsius`:

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

int FahrToCelsius(int fahr);

int tempc;

void main ()
{
  int tempf = 35;

  tempc = FahrToCelsius(tempf);
  printf("%d F -> %d C\n", tempf, tempc);
  printf("%d F -> %d C\n", 12, FahrToCelsius(12));
}

int FahrToCelsius(int fahr)
{
  int celsius;

  celsius = 5*(fahr-32)/9;
  return celsius;
}
```

Programme 45

Dans ce programme, la variable `celsius` ne peut être utilisée que dans la fonction `FahrToCelsius`. L'utiliser dans la fonction `main` produirait une erreur de compilation. De même tenter d'utiliser la variable `tempf` dans la fonction `FahrToCelsius` provoque également une erreur. Par contre, la variable `tempc`, déclarée en dehors de toutes les fonctions, est une variable globale et peut être utilisée à la fois dans `main` et dans `FahrToCelsius`.

Règles de visibilité. La portion d'un programme où une variable existe est appelée *portée* (*scope*) de la variable. A l'intérieur d'une fonction, il est possible d'utiliser des variables locales aussi bien que des variables globales, mais il est recommandé soit de déclarer les variables localement autant que possible, soit de passer les variables globales en paramètres, car ceci évite de manipuler par erreur des variables déclarées dans le programme principal et utilisées également avant et après l'appel à la fonction. Normalement, une fonction ne devrait utiliser pour ses calculs que les valeurs de ses paramètres et de ses variables locales. Cela demande un peu plus de travail au programmeur, mais évite nombre d'erreurs, et rend la fonction réutilisable d'un programme à l'autre.

Deux variables déclarées dans deux fonctions différentes peuvent avoir le même nom: ce seront cependant deux variables différentes. Une variable de programme peut avoir le même nom qu'une variable locale. Dans la fonction, après la déclaration de la variable locale, seule la variable locale reste visible.

Une façon d'éviter toutes ces subtilités est de choisir des noms différents pour toutes les variables de votre programme, quel que soit l'endroit de leur déclaration et de limiter au maximum l'emploi de variables globales. A la rigueur, vous pouvez utiliser la variable `i` comme indice de boucle à plusieurs endroits, mais n'oubliez pas de la déclarer dans toutes les fonctions où vous l'utilisez et surtout ne la déclarez jamais globale.

Ci-dessous vous avez un programme assez simple (programme 46) où le programmeur emploie par inadvertance deux fois la variable `x`, mais ne la déclare qu'une fois globalement. Pouvez-vous expliquer pourquoi ce programme n'affiche aucun ovale alors qu'on pourrait penser qu'il en affiche 40 ?

```
#include <stdio.h>
#include <math.h>
#include "Graphics.h"

int x, y;

int ComputePosition(int abscisse)
{
  int ordonnee = abscisse;

  for(x=1; x<=1000; x++)
    ordonnee = ordonnee * 7 % 400;
  return ordonnee;

}

void main ()
{
  for (x=10; x < 400; x+=10) {
    y = ComputePosition(x);
    DrawOval(x-2,y-2,x+2,y+2);
  }
  getchar();
}
```

Programme 46

L’exemple suivant va vous permettre de vérifier si vous avez compris les règles de visibilité précédentes.

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

float x, fact;

float factorielle(float x)
{
  int i;
  float z = 1.0;

  for (i=2; i<=floor(x); i++) {
    z = z * i;
  }
  x = 0;
  fact = z;
  return fact;
}

void main()
{
  x = 12;
  fact = factorielle(x);
  printf("Factorielle de %1.0f = %1.0f\n", x, fact);

  fact = 4;
  factorielle(4.0);
  printf("Factorielle de 4.0 = %1.0f\n", fact);

}
```

Programme 47

Les noms `x` et `z` déclarés localement dans la fonction factorielle ne sont connus que dans cette fonction. Les variables du programme principal, `x` et `fact`, déclarées après les directives d'inclusion sont connues partout, y compris à l’intérieur de `factorielle`. Cependant le paramètre `x` se comporte comme une variable locale à la fonction factorielle, distincte de la variable du même nom déclarée globalement.

On en tire les conclusions suivantes: l’instruction `printf("%1.0f %1.0f\n",x,fact)` placée dans la fonction affiche la valeur du `x` global. Cette valeur n'est pas changée par l'appel `factorielle(x)` car la variable `x` visible à l'intérieur de la fonction `factorielle` est locale et masque la variable globale du même nom. L'affectation `x=0` n'a donc pas d'incidence sur la variable `x` globale. La valeur du `x` affichée par le premier `printf` du programme principal est donc 12.0.

D’autre part comme `fact` est une variable globale, elle est connue aussi bien à l’intérieur de la fonction `factorielle` qu'à l'intérieur de la fonction `main`. Comme il n'y a pas, à l'intérieur de `factorielle`, de variable de même nom, l'affectation `fact=z;` modifie la variable `fact` globale et le deuxième `printf` affiche donc bien la valeur de la factorielle de 4 et non pas 4.0 comme on pourrait s'y attendre.

8 Adresses et valeurs: pointeurs

Un pointeur est une variable qui contient l'adresse d'une autre variable. Cette notion constitue souvent un obstacle majeur pour les débutants en C. Pourtant, si un usage confus peut effectivement rendre incompréhensible un programme utilisant des pointeurs, le concept est en lui-même très simple à comprendre et repose sur le fonctionnement même des mémoires d'ordinateur.

En C l'utilisation de pointeurs est incontournable car ils sont étroitement liés à la représentation des tableaux et donc des chaînes de caractères. Ainsi, certaines notions qui ont pu vous sembler obscures dans les sections précédentes, comme le fait que l'affectation au moyen du signe = des variables tableaux de caractères ne copiait en fait pas ces chaînes, vont trouver ici des explications lumineuses! N'hésitez pas à relire ce chapitre plusieurs fois.

8.1 Structure de la mémoire d'un ordinateur

Tout d'abord un bref rappel sur le fonctionnement des ordinateurs. Un ordinateur comporte un processeur et de la mémoire. Les deux sont connectés par un bus, c'est-à-dire un ensemble de lignes électriques (figure 6).

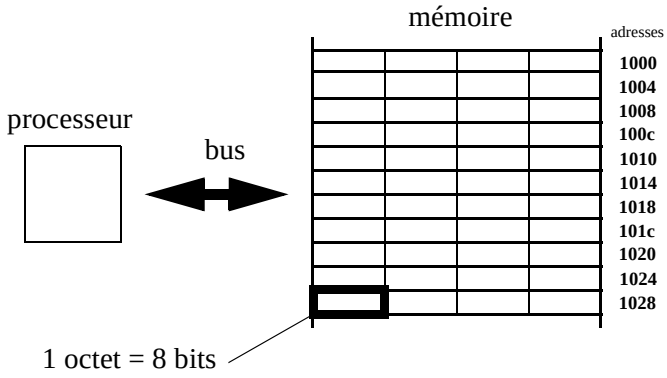


FIGURE 6: Structure d'un ordinateur et de la mémoire

La mémoire d'un ordinateur est en fait un composant électronique (appelé RAM) au fonctionnement relativement simple. Ce composant stocke une série de bits dont on peut écrire la valeur et venir la relire plus tard. Pour des questions de commodité on ne lit/écrit pas ces bits un par un mais par paquets de huit, ce que l'on appelle un *octet* (*byte*). Un unique composant de RAM peut ainsi stocker plusieurs millions d'octets. Quand on veut en lire ou en écrire un il faut naturellement indiquer au composant lequel. Pour cela il suffit de donner son numéro, en effet tous les octets contenus dans la mémoire sont numérotés de 0 jusqu'à n-1 pour une mémoire contenant n octets. Ce numéro permettant d'identifier individuellement chacun des octets de la mémoire est appelé *adresse*. Graphiquement la mémoire d'un ordinateur peut donc être représentée comme une série de cases numérotées de 0 à n-1 contenant chacune un octet, c'est à dire un nombre entier de 0 à 255 (figure 7).

3	255	17	28	32	47	255	0	0	255	28	0
0	1	2	3	4	5	6	7	8	9	10	11

FIGURE 7: Représentation simplifiée du début de la mémoire d'un ordinateur

Grâce aux adresses, un composant de RAM peut donc identifier et lire ou écrire individuellement chacun des octets qu'il contient. Toutefois afin d'aller plus vite et pour manipuler des nombres plus grands (255 ça n'est pas énorme!) les microprocesseurs modernes lisent et écrivent quatre octets consécutifs (32 bits) de la mémoire. C'est pourquoi, pour faciliter la lecture, plutôt que de représenter les octets un à un côte à côte comme dans la figure 7, on a pour habitude dans les schémas, de les représenter groupés par blocs de quatre. Par conséquent les adresses de chaque case évoluent de quatre en quatre comme dans la figure 6. Dans cette figure, une portion de mémoire est représentée couvrant les adresses 1000 à 102b. Par convention, les adresses sont représentées en notation hexadécimale (base 16), en utilisant les chiffres 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f pour les valeurs 0 à 15. Chaque ligne de mémoire dans la figure 6 contient donc 4 octets. En particulier, la première ligne contient les octets d'adresse 1000 (extrême gauche), 1001 (centre gauche), 1002 (centre droite), et 1003 (extrême droite).[1]

En programmation, on devrait donc, pour utiliser des valeurs dans la mémoire procéder à des opérations du genre "ranger la valeur 12 à l'adresse 1036", "prendre la valeur rangée à l'adresse 1024, l'additionner à la valeur

1. Dans les schémas nous utiliserons ces valeurs d'adresse, débutant à la valeur 1000, notez cependant que ce choix est purement arbitraire, généralement les vraies valeurs observées dans les exercices seront beaucoup plus grandes.

rangée à l'adresse 1036 et ranger le résultat de l'opération à l'adresse 1058". Naturellement ceci n'est guère pratique car cela demanderait une sacré gymnastique pour se souvenir de toutes ces adresses, tous les numéros finissant par se ressembler à la longue.

C'est pourquoi le langage C offre des variables. Les variables ne sont ainsi ni plus ni moins que de simples labels arbitraires que l'on donne à des cases mémoires pour éviter d'avoir à utiliser leur adresse numérique. On peut ainsi utiliser des noms plus faciles à retenir plutôt que des adresses numériques. Lorsque vous déclarez des variables dans un programme C, le compilateur choisit lui-même des emplacements disponibles dans la mémoire et établit une correspondance entre les noms de variables choisis et les adresses de ces emplacements. Considérons le programme suivant:

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

void main()
{
  int x, i;
  float r;
  short int j,k;
  char str[24];

  x = 3;
  strcpy(str,"Hello");
  x = x + 1;
}
```

Programme 48

En supposant qu'un `float` prend 8 octets de place mémoire, un `int` 4 octets, un `short` 2 octets, et une chaîne caractères, n octets, le compilateur pourrait organiser les données en mémoire comme suit:

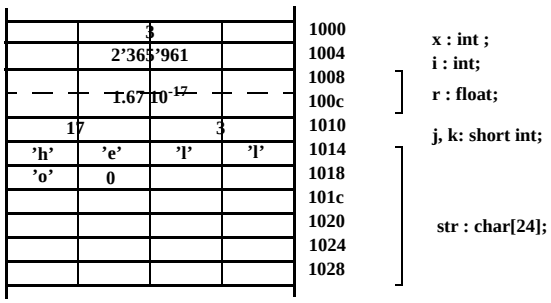


FIGURE 8: Organisation de variables en mémoire

Au passage, notez le 0 à l'adresse 1019 qui indique la fin de la chaîne à l'intérieur du tableau de 24 caractères.

Le compilateur retient les adresses de toutes les variables. Quand on utilise la variable `x` dans le programme, en fait on utilise l'adresse 1000 (et implicitement les 3 suivantes, 1001, 1002 et 1003 si un `int` occupe bien 32 bits comme nous l'avons supposé), on dit que l'adresse de la variable `x` est 1000. Lorsque l'on écrit l'instruction x = 3, ce que le programme fait à l'exécution, c'est aller écrire à l'adresse 1000 en mémoire la valeur 3 codée en binaire sur 4 octets (00000000 00000000 00000000 00000011). Lorsque vous écrivez l'instruction `x=x+1`, ce que le programme fait à l'exécution, c'est aller prendre la valeur de 4 octets stockée à l'adresse 1000, la donner au processeur qui lui ajoute la valeur 1, et la réécrire à la même adresse 1000.

Ce modèle d'exécution est extrêmement simple, mais très général. C'est ce qui fait la puissance des ordinateurs. La chose importante à retenir donc est que chaque variable que vous déclarez est caractérisée par deux choses: son adresse en mémoire et sa valeur.

8.2 Les pointeurs

Comme indiqué précédemment, les adresses des octets en mémoire et par la même, les adresses des variables, sont de simples nombres entiers. Rien n'empêche donc d'affecter ces adresses à une autre variable. Considérons la figure 8, on y remarque que l'adresse de la variable `i` est 1004, nous pouvons tout à fait stocker cette valeur 1004 dans une autre variable, c'est-à-dire dans un autre emplacement de la mémoire. Une telle variable numérique qui contient une valeur qui n'est pas réellement utile en soi mais se trouve être l'adresse d'une autre variable est ce que l'on appelle un *pointeur* car elle indique (pointe) l'emplacement d'une autre variable dans la mémoire.

Le langage C dispose d'un opérateur pour prendre l'adresse d'une variable, l'opérateur unaire `&`. Ainsi dans la situation de la figure 8 toujours, l'expression

```
p=&i;
```

met dans la variable `p` la valeur 1004, adresse de l'emplacement mémoire correspondant à la variable `i`. On dit alors que `p` pointe sur `i` ou que `p` est un pointeur sur `i`.

Un pointeur est donc une variable contenant une adresse. Comme une adresse est un nombre entier, on pourrait penser que les pointeurs sont des variables de type `int` tout simplement. En fait pas exactement, le langage C distingue les pointeurs suivant le type de la variable qu'ils pointent. Ainsi `p` qui contient l'adresse de la variable `i` doit être de type “pointeur sur `int`”. Le langage C utilise la notation suivante pour déclarer une variable d'un tel type:

```
int *p;
```

Le langage C offre également un opérateur permettant d'accéder à la valeur stockée dans une case mémoire dont on connaît l'adresse, il s'agit de l'opérateur unaire `*`. Dans notre exemple, après l'instruction `p=&i`, la variable `p` contient la valeur 1004 représentant l'adresse en mémoire où se trouve stocké un entier (en l'occurrence la valeur de `i`). Dès lors, la construction `*p` a pour valeur l'entier stocké à l'adresse 1004, c'est-à-dire 2 365 961. Ainsi, si `j` est de type `int`, la construction suivante:

```
j = *p;
```

met la valeur 2 365 961 dans la variable `j`. Cette opération consistant à prendre la valeur de la case mémoire pointée par un pointeur constitue un *déréférencement* ou *indirection*. Mais la construction `*p` peut également être utilisée pour ranger une nouvelle valeur dans la case mémoire dont l'adresse est contenue dans `p`. Ainsi l'instruction:

```
*p=12;
```

range la valeur 12 dans la case mémoire d'adresse 1004, valeur de `p`. Or la variable `i` est rangée à cet endroit de la mémoire aussi, donc cette opération a pour effet de changer la valeur de la variable `i` également qui devient égale à 12.

A beaucoup d'égards, comme ils contiennent des adresses qui ne sont en fait que des nombres, les pointeurs se comportent comme de simples variables numériques. Ainsi si l'on définit un autre pointeur sur `int`: `int *q;` l'affectation `q=p;` est légale. Elle a pour effet de fixer la valeur de `q` à 1004. Donc `q` se retrouve contenir l'adresse de `i`, tout comme `p`. `q` devient ainsi un autre pointeur sur `i` et toute utilisation des expressions `*q` et `*p` est donc rigoureusement équivalente.

Considérons le programme suivant qui résume ces notions:

Adresses:

```
void main()
{
  int v=12;
  int u=10;
  int *vP; /*pointeur sur int*/

  vP = &v; /*affectation du pointeur */
  u = *vP;
  printf("u=%d v=%d\n",u,v);
  *vP = 25;
  printf("u=%d v=%d\n",u,v);
}
```

Programme 49

L'organisation des données en mémoire est la suivante:

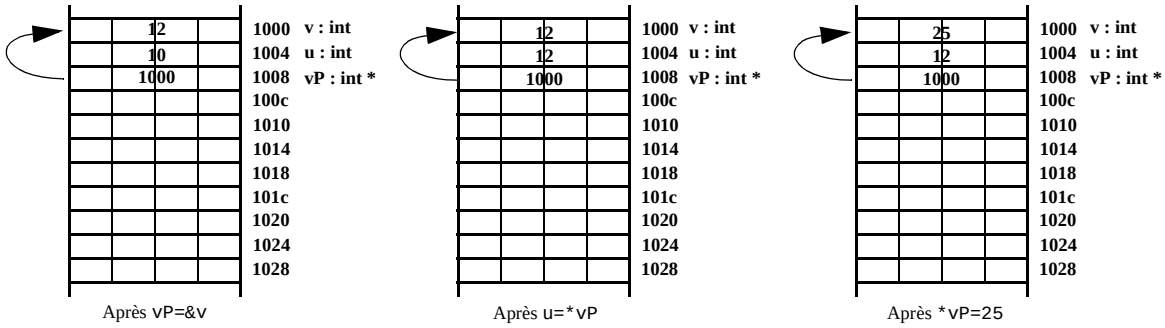


FIGURE 9: Organisation de la mémoire (pointeurs)

On a ici symbolisé par une flèche le fait que la variable `vP` pointe vers la variable `v`. Immédiatement après l'instruction `vP=&v`, `u` et `v` contiennent leur valeur d'initialisation, soit respectivement 12 et 10. Ces deux variables sont rangées en mémoire aux adresses 1000 et 1004 respectivement. Suite à l'instruction `vP=&v`, la variable `vP`, stockée en mémoire à l'adresse 1008, contient la valeur 1000, adresse de `u`. L'affectation `u=*vP` met dans la variable `u`, la valeur pointée par `vP`, c'est-à-dire la valeur se trouvant en mémoire à l'adresse 1000, en l'occurrence 12. La première instruction `printf` affiche donc tout naturellement les valeurs de u et v: 12 et 12. L'instruction `*vP=25` demande de mettre la valeur 25 dans la case mémoire pointée par `vP`, c'est-à-dire d'adresse 1000. Comme la variable `v` est précisément rangée à l'adresse 1000, cette instruction a pour effet de changer la valeur de `v` et le deuxième `printf` affiche de ce fait les valeurs de `u` et `v`: 12 et 25.

A chaque type prédéfini du langage ou bien défini par le programmeur (voir l'instruction `typedef`, section 10.1) on associe un type pointeur. On peut ainsi définir des variables de type "pointeur sur `float`", des variables de type "pointeur sur `char`", etc... De tels types de pointeurs sont incompatibles, bien que tous soient en réalité des adresses de variables, donc contiennent en fait une valeur numérique entière. Il n'est pas possible de mettre telle quelle l'adresse d'une variable de type `int` dans un pointeur sur `float`. Ceci est dû au fait que lors d'un déréférencement, c'est-à-dire quand on examine le contenu d'un emplacement de mémoire pointé par un pointeur, au moyen de la construction `*ptr`, l'interprétation faite des nombres contenus dans la mémoire varie suivant le type considéré. Ainsi si `ip` est un pointeur sur un entier: `int *ip`, l'instruction `i=*ip` doit aller examiner le contenu de la mémoire à l'adresse contenue dans `ip`. Comme `ip` est un pointeur sur un `int` et qu'un `int` occupe 32 bits, on sait que l'on doit considérer 4 octets consécutifs à partir de l'adresse contenue dans `ip` pour déterminer la valeur de `*ip`. Maintenant si `sp` est un pointeur sur un `short`: `short int *sp`, l'instruction `s=*sp` doit de même aller examiner le contenu de la mémoire à l'adresse contenue dans `sp`. Mais comme `sp` est un pointeur sur un `short` et qu'un `short` n'occupe que deux octets en mémoire, il suffit d'examiner deux octets et non plus quatre pour déterminer la valeur de `*sp`. On voit ainsi que, bien que `ip` et `sp`

contiennent tous deux des adresses numériques en mémoire analogues, ils sont en fait de types différents car l'interprétation qui doit être faite de `*ip` et `*sp` respectivement est différente. C'est pourquoi le programme suivant provoque un avertissement du compilateur:

```
main()
{
  int *ip;
  short s;

  ip = &s;   /* Avertissement du compilateur !!! */
}
```

Programme 50

En effet `&s` est une expression de type “pointeur sur short” (`short int *`) et on tente de l'affecter à une variable de type “pointeur sur int” (`int *`).

Pointeur générique. Dans certaines situations il est nécessaire de conserver et manipuler une adresse mémoire sans nécessairement savoir ou donner d'indication sur le type des données qui sont stockées à cet endroit. Le C dispose pour de tels cas du type *pointeur générique*. On déclare un tel pointeur de la façon suivante:

```
void *ptr;
```

Un tel pointeur ne peut pas être déréférencé, c'est-à-dire qu'il est illégal d'utiliser l'expression `*ptr`, ce qui s'explique simplement par le fait que l'on ne dispose pas d'information sur le type de données contenu en mémoire à l'adresse indiquée par `ptr` (on ne sait donc pas si on doit considérer un, deux, quatre octets ou plus pour déterminer la valeur de `*ptr`). Si l'on veut pouvoir déréférencer un pointeur générique, il faut donner une indication sur les données contenues en mémoire à l'adresse indiquée par le pointeur en utilisant une conversion explicite du type du pointeur (*cast*). Par exemple: `(int *)ptr` convertit le pointeur générique en pointeur sur `int`. Nous reviendrons sur les pointeurs génériques dans la section consacrée à l'allocation dynamique de la mémoire (section 8.6).

Pointeur NULL. Avant d'être initialisée, une variable locale de type pointeur, comme toute variable locale non initialisée, contient une valeur indéterminée. S'il n'y prend garde, le programmeur peut tenter d'utiliser une telle valeur qui pointe une case mémoire quelconque, cela conduit le plus souvent à un plantage immédiat du programme. Pour éviter ce problème, il est courant d'affecter à une variable de type pointeur la constante NULL qui vaut zéro. Cette valeur particulière indique de façon explicite que le pointeur n'est pas initialisé et ne contient pas d'adresse utilisable.

8.3 Les tableaux

Comme indiqué en début de chapitre, pointeurs et tableaux sont étroitement liés. Nous avons signalé le fait que le langage C établit une correspondance entre tous les noms symboliques de variables et des adresses en mémoire où sont rangées les valeurs de ces variables. Ainsi dans le cas de la figure 8, le nom symbolique `i` est équivalent en fait à l'adresse 1004, le nom `x` à l'adresse 1000. Le contenu de la mémoire à ces adresses est la valeur des variables `i` et `x`. Qu'en est-il pour un tableau tel que:

```
int t[5];
```

A quoi correspond le nom symbolique `t`? A quelle adresse en mémoire est-il associé? Le langage C ne reconnaissant que des types de base proches de ceux que manipule le microprocesseur (donc essentiellement des nombres), il n'est pas en mesure d'associer le nom `t` aux 20 octets qu'occupe effectivement en mémoire un

tableau déclaré de cette façon. En fait C associe au nom `t` simplement l'adresse où commence le tableau en mémoire. Considérons le programme suivant:

```
#include <stdio.h>
main()
{
  int t[5] = {31, 14, 21, 12, 24};
  int *ip;

  printf("t=%x\n", t);          /* Affiche la valeur de t en hexa */
  printf("&t[0]=%x\n", &t[0]);  /* Affiche l'adresse du 1er element du tableau */

  printf("*t=%d\n", *t);          /* Affiche la valeur de *t */
  printf("t[0]=%d\n",t[0]);     /* Affiche la valeur de t[0] */

  ip = t;
  *ip = 17;
  printf("t[0]=%d\n", t[0]);
}
```

Programme 51

Les trois premières lignes de la fonction `main` déclarent trois variables `t`, `ip` et `i`. Voici le contenu de la mémoire juste après l'exécution de ces 3 lignes:

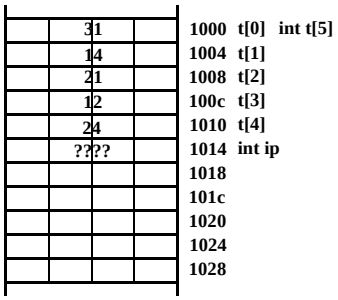


FIGURE 10: Organisation de la mémoire (programme 51, initialisations)

La déclaration `int t[5]` définit un tableau de 5 `int`s, c'est-à-dire un bloc de 5 entiers consécutifs en mémoire notés `t[0]`, `t[1]`, ...`t[4]`. Les deux premiers appels à la fonction `printf` affichent les valeurs des variables `t` et `&t[0]` en hexadécimal. On constate en exécutant ce programme que ces valeurs sont les mêmes. La valeur de `t` est l'adresse du premier élément du tableau, `t[0]`. Cette remarque est d'une extrême importance.

La valeur d'une variable de type tableau est l'adresse du premier élément du tableau

`t` se comporte en quelque sorte comme un pointeur sur le premier élément du tableau. De ce fait, l'expression `*t` est légale et a pour valeur, la valeur du premier élément du tableau `t[0]`. C'est ce que l'on constate grâce aux messages affichés par les deux instructions `printf` suivantes du programme 51**.**

Comme la valeur de `t` est l'adresse, 1000, du premier élément du tableau et que le tableau contient des `int`s, l'affectation `ip=t` est légale et place la valeur 1000 à l'adresse 1014 associée à `ip`, faisant de ce fait pointer `ip` sur le premier élément du tableau, `t[0]`. La mémoire est alors dans l'état suivant:

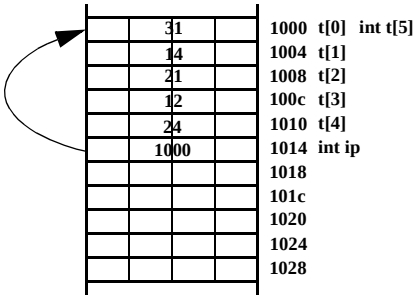


FIGURE 11: Organisation de la mémoire (programme 51, affectation de ip)

L'affectation suivante, `*ip=17` a pour effet de placer la valeur 17 à l'adresse pointée par `ip`, en l'occurrence 1000, c'est-à-dire que la valeur de `t[0]` est changée, ce que confirme le `printf` suivant.

Par de nombreux aspects donc, le nom de tableau, `t`, se comporte comme un pointeur. Il existe toutefois une différence essentielle: un nom de tableau n'est pas une variable au sens propre et on ne peut donc en changer la valeur. Des instructions telles que `t=&i` ou `t++` sont signalées comme des erreurs par le compilateur. Ceci explique que l'on ne puisse utiliser simplement l'opérateur = pour copier un tableau dans un autre, cela ne ferait pas de sens.

Tableaux multi-dimensionnels. Les tableaux multi-dimensionnels sont en tout point semblables aux tableaux mono-dimensionnels évoqués précédemment. Là encore les différents éléments du tableau sont disposés dans des espaces successifs en mémoire et le nom symbolique associé au tableau a en fait pour valeur l'adresse du premier élément du tableau. Prenons le cas du programme suivant:

```
#include <stdio.h>

void main()
{
  int i,j;
  int m[4][4];

  for(i=0;i<4;i++)
    for(j=0;j<4;j++)
      m[i][j] = i + j;

  for(i=0;i<4;i++) {
    for(j=0;j<4;j++)
      printf("%d ",m[i][j]);
    printf("\n");
  }
}
```

Programme 52

L'organisation des variables en mémoire pourrait être la suivante. La variable `i` de type `int` occupe 4 octets à partir de l'adresse 1000. La variable `j` occupe 4 octets aux adresses à partir de l'adresse 1004. La matrice `m[4][4]` occupe 64 octets (4x4x4 octets), aux adresses 1008 à 1048. La variable `m` prise isolément (sans crochets) a pour valeur l'adresse du premier élément de la matrice, c'est-à-dire 1008. Les éléments d'une matrice sont organisés par ligne en mémoire. Cela veut dire que l'élément `m[0][0]` occupe 4 octets aux adresses 1008, 1009, 100A, 100B, l'élément `m[0][1]` occupe 4 octets aux adresses 100C, 100D, 100E, 100F, l'élément `m[0][2]` occupe 4 octets aux adresses 1010, 1011, 1012, 1013; l'élément `m[0][3]` occupe 4 octets aux adresses 1014, 1015, 1016, 1017; l'élément `m[1][0]` occupe 4 octets aux adresses 1018, 1019, 101A, 101B et ainsi de suite. Alternativement, on peut dire que l'adresse de `m[0][0]` est 1008, l'adresse de `m[0][1]` est 100C,

l'adresse de `m[0][2]` est 1010, l'adresse de `m[0][3]` est 1014; l'adresse de `m[1][0]` est 1018 et ainsi de suite (figure 12).

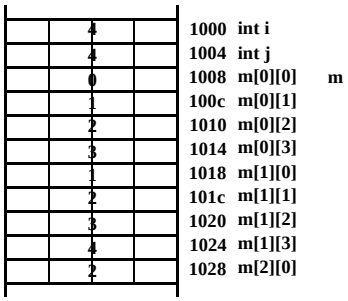


FIGURE 12: Organisation de la mémoire (tableau bidimensionnel)

8.4 Arithmétique des pointeurs

Les pointeurs étant des variables contenant des adresses numériques donc en fait des nombres entiers, on peut leur appliquer des opérations arithmétiques, notamment incrémentation et décrémentation. Toutefois il y a quelques subtilités de taille qui distinguent l'arithmétique des pointeurs de celle des simples entiers.

Considérons le programme suivant:

```
#include <stdio.h>
main()
{
  int t[5] = {31, 14, 21, 12, 24};
  int *ip, *ipbis;

  ip = t;
  ipbis = t+1;
  printf("%d %d\n",*ip,*ipbis);
  printf("ip=%x\nipbis=%x\n",ip,ipbis);
}
```

Programme 53

Après son exécution la mémoire se trouve dans l'état suivant:

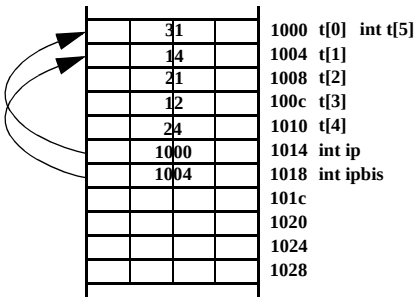


FIGURE 13: Arithmétique des pointeurs

L'affectation `ip=t`, place la valeur 1000 dans `ip`, faisant pointer cette variable vers le premier élément du tableau `t`, `t[0]`. Par définition, le langage C assure que `ip+1` pointe vers le deuxième élément du tableau, `ip+2` pointe vers le troisième et ainsi de suite. Ainsi l'instruction `ipbis=ip+1` place dans `ipbis` l'adresse du deuxième élément du tableau, `t[1]`, c'est-à-dire la valeur 1004 et non pas 1000+1=1001 comme on aurait pu le croire à première vue. Cette affirmation est d'ailleurs confirmée par le résultat des deux `printf`s.

Quel que soit le type pointé, si `p` est un pointeur sur un élément d'un tableau, alors `p+1` est un pointeur sur l'élément suivant, `p+i` est un pointeur sur le `i`ème élément suivant.

Dès lors on peut déduire un certain nombre d'équivalences d'écriture. En reprenant les définitions du programme 53, comme `ipbis` est un pointeur sur `t[1]`, alors `*ipbis`, qui dénote la valeur se trouvant dans la case mémoire pointée par `ipbis`, est équivalent à `t[1]`. En généralisant, si `i` est un entier entre 0 et 4 alors `t+i` est équivalent à `&t[i]` et donc `*(t+i)` est équivalent à `t[i]`.

Une fréquente source d'erreur chez les débutants en C consiste à confondre *valeur du pointeur* et *valeur pointée par le pointeur*. Dans la situation de la figure 13, la valeur du pointeur `ip` est 1000, celle du pointeur `ipbis` est 1004. La valeur pointée par `ip` (notée `*ip`) est 31, celle pointée par `ipbis` (notée `*ipbis`) est 14.

Il y a de ce fait une différence fondamentale entre les deux expressions `ip=ipbis` et `*ip=*ipbis`. La première fait passer la valeur de `ip` de 1000 à 1004 qui est la valeur de `ipbis`. `ip` se retrouve ainsi pointant sur le même emplacement mémoire que `ipbis`. La deuxième expression, `*ip=*ipbis`, fait passer la valeur pointée par `ip` de 31 à 14 qui est la valeur pointée par `ipbis`. Le configuration résultant de ces deux expressions est illustrée dans la figure suivante:

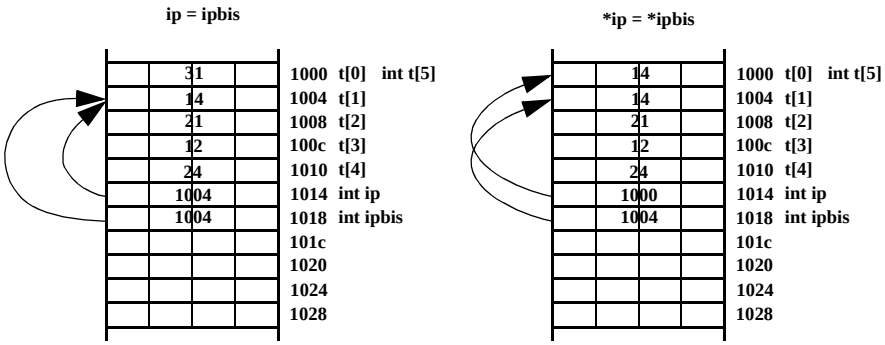


FIGURE 14: Valeur de pointeur et valeur pointée

8.5 Utilisation de pointeurs comme arguments de fonctions

Comme il a été signalé dans la section 7.5, lors d'un appel de fonction les valeurs des variables sont copiées pour être passées à la fonction (passage par valeur). Dès lors il ne semble pas possible de modifier une variable depuis l'intérieur d'une fonction. Considérons l'exemple suivant:

```
#include <stdio.h>

void incremente(int e)
{
  e = e+1;
}

main()
{
  int i = 12;
  incremente(i);
  printf("i=%d\n",i);
}
```

Programme 54

En raison du passage par valeur des paramètres, ce programme affiche la valeur 12 et non pas 13. Lors de l'appel de la fonction `incremente(i)`, c'est la valeur de `i`, c'est-à-dire le nombre 12, qui est copiée dans le paramètre `e` pour la durée de l'exécution de la fonction `incremente`. Celle-ci incrémente la variable `e` et la

fait donc passer à 13, mais comme cette variable n'existe que dans le corps de la fonction `incremente` (voir section 7.5 sur la localité des variables), cette valeur est perdue lors du retour dans la fonction `main`.

La variable `i` est locale à `main` et n'est donc pas visible depuis `incremente`. Il existe néanmoins un moyen pour en modifier le contenu depuis la fonction `incremente`, il faut pour cela utiliser un pointeur. L'idée consiste à transmettre à la fonction `incremente`, plutôt qu'une valeur à incrémenter comme nous l'avons fait jusqu'ici, l'adresse en mémoire où se trouve l'entier à incrémenter. En quelque sorte, jusqu'à maintenant le corps de la fonction faisait "ajoute un au nombre entier qui est donné" et ce nombre était perdu. Désormais il faut qu'elle fasse "ajoute un à l'entier se trouvant à tel endroit donné dans la mémoire". Si cet endroit de la mémoire correspond à une variable alors la valeur de cette variable sera changée, même si elle est extérieure à la fonction! Le programme précédent est modifié comme suit:

```
#include <stdio.h>

void incremente(int *e)
{
  *e = *e+1;
}

main()
{
  int i = 12;
  incremente(&i);
  printf("i=%d\n",i);
}
```

Programme 55

On remarque dans cette nouvelle version que la fonction `incremente` attend désormais comme paramètre, non plus un simple nombre entier, mais plutôt un pointeur sur un nombre entier (`int *e`), c'est-à-dire l'adresse d'une case mémoire où se trouve un nombre entier. Ce que fait désormais la fonction c'est d'aller consulter cet emplacement de la mémoire pour relever l'entier qui s'y trouve contenu, y ajouter un et replacer cette nouvelle valeur au même endroit de la mémoire (`*e=*e+1`).

Considérons la fonction `main` maintenant. La première ligne définit la variable `i`. Ceci revient pour le compilateur à réserver 4 octets en mémoire, y associer le nom symbolique i et à ranger dans cet emplacement la valeur 12. Admettons, pour fixer les idées, que ces 4 octets soient réservés à partir de l'adresse 1032. Vient alors l'appel `incremente(&i)`. L'expression `&i` a pour valeur 1032, c'est-à-dire l'adresse de l'emplacement où se trouve rangée la variable `i` en mémoire. Comme le C transmet les paramètres de fonction par valeur, ce nombre, 1032, est copié momentanément dans la variable `e` pour la durée de l'exécution de la fonction `incremente`. Cette fonction est alors exécutée, `e` valant 1032, c'est-à-dire pointant sur `i`. Comme décrit précédemment, l'expression `*e=*e+1` va alors chercher, la valeur se trouvant en mémoire à l'adresse 1032. Cette valeur est 12 naturellement. Cette valeur est incrémentée et le résultat, 13, rangé à l'adresse 1032. Ce faisant, la valeur de la variable `i` de la fonction `main` est bien changée et devient 13, ce que confirme le `printf`.

Ainsi programmée, la fonction `incremente` est à même de changer le contenu d'une variable externe. Ce mécanisme consistant à passer comme argument de fonction non pas la valeur d'une variable mais son adresse est très souvent utilisé pour plusieurs raisons:

- comme les paramètres sont passés par valeur, c'est-à-dire copiés pour être utilisés localement par la fonction, le temps de copie peut devenir non négligeable si le type de l'objet sur lequel la fonction travaille est grand. Dans ce cas il est plus judicieux d'éviter la copie de l'objet et de passer simplement à la fonction un pointeur dessus, c'est-à-dire l'adresse en mémoire où l'objet peut être trouvé
- une fonction ne peut retourner qu'une valeur par la construction `return` (même si cette valeur peut être complexe). Or, dans certains cas, on peut souhaiter plusieurs valeurs en retour. Ainsi la fonction `FahrToCelsius` du programme 44 n'a besoin de renvoyer qu'une température et peut donc être codée sous la forme d'une fonction retournant un `int`. Imaginons par contre une fonction convertissant des

coordonnées cartésiennes du plan (x, y) en coordonnées polaires (r, θ). Une telle fonction a besoin de retourner deux valeurs. Pour cela on peut lui passer comme paramètre l'adresse de deux variables où la fonction pourra ranger les valeurs calculées de r et θ

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

void CartToSpher(float x, float y, float *r, float *theta)
{
  *r = sqrt(x*x + y*y);
  *theta = atan(y/x);
}

void main ()
{
  float x1,y1;
  float r1,theta1;

  x1 = 1;
  y1 = 1;
  CartToSpher(x1,y1,&r1,&theta1);
  printf("r1 = %5.3f theta1 = %5.3f (radians)\n",r1,theta1);
}
```

Programme 56

Le programme 56 illustre ce dernier cas. La fonction `CartToSpher` a pour paramètres deux réels puis deux adresses de réels où seront stockées les valeurs de coordonnées réelles calculées sur le même principe que la fonction `incremente` précédente.

Exercice 66. *Ecrire une fonction qui permet d'échanger les valeurs de deux variables passées en paramètre*

Notons pour terminer que nous avons déjà rencontré une fonction modifiant des variables externes en utilisant pour cela un passage de paramètres par adresse: la fonction `scanf`. Grâce à ce qui précède on comprend pourquoi tous les arguments de cette fonction doivent être des pointeurs. On comprend également pourquoi dans le cas de variables simples on utilise l'opérateur & pour en obtenir l'adresse alors que cela n'est pas nécessaire dans le cas de tableaux puisque leur nom représente déjà un pointeur.

Passage de tableaux en paramètre. Comme expliqué à la section 8.3, les variables de type tableau ont en fait pour valeur l'adresse du premier élément du tableau et, de ce fait, se comportent comme des pointeurs. Donc dire que l'on passe une variable de type tableau à une fonction est en fait un **abus de langage**, on passe en fait l'adresse de début de ce tableau et non pas une copie du tableau. Cela peut être vu comme une entorse au principe de passage des paramètres par valeur et il faut donc y faire attention.

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

void Vecteur2DCarre(int vect[])
{
  vect[0] = vect[0] * vect[0];
  vect[1] = vect[1] * vect[1];
}

void main()
{
  int vecteur[2] = {3, 5};
  Vecteur2DCarre(vecteur);
  printf("Vecteur[0] : %d\n",vecteur[0]);
  printf("Vecteur[1] : %d\n",vecteur[1]);
}
```

Programme 57

Le programme 57 montre bien que le tableau `vecteur`, local à `main`, passé à la fonction `Vecteur2DCarre` est bien modifié par cette dernière. L'interface de la fonction `Vecteur2DCarre` aurait également pu s´écrire

```
void Vecteur2DCarre(int *vect)
```

en tant que paramètres formels de fonction, les notations *`type *param`* et *`type param`*`[]` sont en effet équivalentes. La notation *`type param`*`[]` a toutefois l'avantage d'être plus claire dans ce cas car elle indique qu'il s'agit bien de l'adresse d'un tableau et non pas de celle d'un simple entier que l'on passe à la fonction.

On peut noter au passage qu'aucune information sur la taille du tableau n'est passée à la fonction `Vecteur2DCarre`. De l'intérieur de la fonction, rien ne garantit que le tableau effectivement passé à la fonction est bien de taille 2 comme la fonction s'y attend et rien ne permet de le vérifier. Seule une bonne discipline du programmeur permet de ne pas faire d'erreur dans ce cas. Autant dire que cette situation est une cause potentielle de nombreuses erreurs.

La fonction `Vecteur2DCarre` a une sémantique qui indique qu'elle travaille sur des tableaux de taille fixe deux. Par contre d'autres fonctions peuvent avoir besoin de travailler sur des tableaux de taille quelconque. Imaginons une fonction `SommeVecteur` calculant la somme des éléments d'un tableau d'entiers de taille variable. Le premier paramètre d'une telle fonction serait un tableau d'entiers `int t[]`. A l'intérieur on additionnerait les éléments du tableau `t[0]`, `t[1]`,... mais à quel indice s'arrêter? En passant `t` comme paramètre à la fonction on lui indique juste le début de la zone de mémoire où sont rangées les valeurs des différents entiers du tableau, on ne lui indique pas combien il y en a ou bien où s'arrête le tableau. On peut trouver plusieurs solutions à ce problème. La plus simple consiste à rajouter un paramètre à la fonction permettant d'indiquer le nombre d'éléments du tableau elle est illustrée dans le programme 58.

```
#include <stdio.h>

int SommeVecteur(int t[], int taille)
{
  int i, somme=0;
  for (i=0; i<taille; i++) {
    somme += t[i];
  }
  return sommae;
}

main()
{
  int vect[5] = {2,3,5,3,2};
  printf("Somme=%d\n", SommeVecteur(vect, 5));
}
```

Programme 58

La seconde consiste à choisir une valeur particulière comme marquant la fin du tableau. On peut par exemple décider que la valeur 0 n'est pas un élément normal dans un tableau passé à la fonction et que cette valeur mar-

que au contraire la fin de ce tableau. Dans ce cas il suffit de sommer les éléments du tableau jusqu'à ce qu'un zéro soit rencontré. Cette solution est illustrée dans le programme 59.

```
#include <stdio.h>

int SommeVecteur(int t[])
{
  int i, somme=0;
  for (i=0; t[i]!=0; i++) {
    somme += t[i];
  }
  return somme;
}

main()
{
  int vect[6] = {2,3,5,3,2,0};
  printf("Somme=%d\n", SommeVecteur(vect));
}
```

Programme 59

En pratique on utiliserait plutôt pour cet exemple, la première méthode. La deuxième n'est pas complètement dénuée de sens toutefois puisque c'est celle qui est utilisée pour la manipulation des chaînes de caractères comme le montre le paragraphe suivant.

Exercice 67. *Ecrivez et testez une fonction qui calcule la moyenne d'un vecteur de taille quelconque.*

Exercice 68. *Ecrivez et testez une fonction qui calcule la moyenne et la variance d'un vecteur de taille quelconque.*

Exercice 69. *Ecrivez une fonction qui trouve l'indice des éléments minimum et maximum dans un vecteur de taille quelconque.*

Exercice 70. *Ecrire un programme qui calcule la valeur d'un polynôme de degré N, connaissant ses N+1 coefficients et l'abscisse x (voir exercice 63).*

Passage de pointeurs et tableaux de caractères en paramètre. La section 8.3 a montré combien les tableaux et pointeurs sont étroitement liés. En langage C, les chaînes de caractères sont manipulées sous forme de tableaux de caractères et il existe donc là encore une liaison très forte avec la notion de pointeurs. Les pointeurs de caractères sont, de très loin, les plus courants dans les programmes écrits en langage C.

Comme l'indiquait la section 6.12, il n'existe pas de type primitif "chaîne de caractères", les chaînes ne sont manipulées que sous la forme de tableaux de caractères. A l'intérieur d'un tel tableau un caractère nul, `'\0'`, marque la fin de la chaîne. Tout ce qui a été dit concernant le passage de tableaux comme arguments de fonctions s'applique donc aux fonctions recevant des chaînes de caractères comme paramètres. Là encore, dire que l'on passe une chaîne de caractères comme paramètre à une fonction est un **abus de langage** car en réalité il faut garder à l'esprit que les chaînes de caractères ne sont passées aux fonctions que sous forme de pointeur vers leur premier caractère. Aucune copie n'est faite implicitement (si ce n'est celle de la valeur du pointeur) et donc toute modification dans une fonction d'une chaîne passée en paramètre entraîne bien la modification de la chaîne originale.

Dès lors, appliquons tout ceci à la rédaction d'une fonction permettant de changer les `o` en `a` dans une chaîne de caractères passée en paramètre:

```
#include <stdio.h>

void ChangeOA(char s[])
{
  int i=0;
  while (s[i] != 0) {
    if (s[i]=='o')
      s[i] = 'a';
    i++;
  }
}

main()
{
  char chaine[]="Hello world !";
  ChangeOA(chaine);
  printf("%s\n", chaine);
}
```

Programme 60

Dans cet exemple un tableau de caractères `chaine` est alloué et initialisé avec la chaîne `"Hello world !"`. Pour fixer les idées disons que ce tableau est située dans une zone mémoire commençant à l'adresse 1036. Lors de l'appel de la fonction `ChangeOA` cette valeur, 1036, est affectée au paramètre `s` qui se trouve ainsi pointant sur le début de la chaîne `chaine` en mémoire. Cette situation est illustrée par la figure 15. A l'intérieur de la

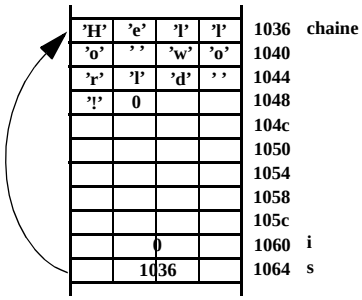


FIGURE 15: Organisation de la mémoire (appel de fonction et chaîne de caractères)

fonction `ChangeOA`, toute expression de la forme `s[i]` fait donc référence à un caractère de `chaine` et non pas une quelconque copie. Toute affectation dans `s[i]` change donc bien `chaine` comme le confirme le `printf`. Par contre tout changement de la valeur de `s` elle même est sans effet sur la chaîne. Tout changement de la valeur de `s` ne fait que changer la case mémoire pointée mais pas la valeur contenue dans cette case mémoire. Grâce à cette remarque et en utilisant les propriété de l'arithmétique des pointeurs évoquées à la section 8.4, on peut éviter l'utilisation de la variable additionnelle `i` dans la fonction `ChangeOA`: il suffit de changer la valeur de `s` pour lui faire pointer successivement un par un les caractères de la chaîne en s'arrêtant au zéro final, c'est ce que fait la version améliorée suivante:

```
void ChangeOA(char s[])
{
  while (*s != 0) {
    if (*s == 'o')
      *s = 'a';
    s++;
  }
}
```

Programme 61 (extrait)

Exercice 71. *Ecrire la fonction strlen(char s[]) qui retourne la longueur de la chaîne de caractères s*

Exercice 72. *Ecrire la fonction strcpy(char dest[], char src[]) qui copie une chaîne src (y compris le '\0' final) dans le tableau de caractères dest que l'on supposera de taille suffisante*

Exercice 73. *Ecrire la fonction strncpy(char dest[], char src[], int n) qui copie n caractères de la chaîne src dans le tableau de caractères dest en tronquant src si elle est trop longue ou en remplissant avec des zéros si elle est trop courte*

8.6 Pointeurs et allocation dynamique de variables

Une autre situation très courante où les pointeurs sont utilisés concerne ce que l'on appelle l'allocation dynamique d'espace mémoire. Il existe en effet des situations où l'on ne sait pas, au moment où l'on écrit un programme, quelle taille il va falloir donner à un tableau. Imaginons par exemple, un programme qui lise un texte depuis le disque dur pour le mettre dans un tableau de caractères en mémoire afin de pouvoir le manipuler. Au moment où l'on écrit le programme, on ne sait pas quelle va être la taille nécessaire pour ce tableau, il est donc difficile de le déclarer comme nous l'avons fait jusqu'ici. Une solution consiste bien sûr à se donner une taille très grande quitte à n'en utiliser en fait qu'une toute petite partie, mais cette solution n'est pas bonne car elle provoque un gaspillage de mémoire qui est une ressource toujours limitée.

Le langage C offre une solution à ce problème au travers des deux fonctions standard `malloc()` et `free()` déclarées dans `stdlib.h`. Le programme suivant montre l'utilisation de ces deux fonctions. Il demande à l'utilisateur combien il souhaite rentrer de nombres. Ensuite on lui demande de rentrer ces nombres un par un. On affiche ensuite la série de nombres entrés par l'utilisateur.

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

void main()
{
  int i, n;
  int *ptr;

  printf("Combien d'entiers voulez vous rentrer ? ");
  scanf("%d",&n);
  ptr = malloc(n*sizeof(int));
  if (ptr == NULL) {
    printf("ERREUR: allocation du tableau impossible\n");
    exit(1);
  }
  for (i=0; i<n; i++) {
    printf("Entrez l'entier d'indice %2d: ", i);
    scanf("%d",&ptr[i]);
  }
  printf("\nContenu du tableau: \n", ptr);
  for (i=0; i<n; i++) {
    printf("%d ", ptr[i]);
  }
  free(ptr);
}
```

Programme 62

Le nombre de nombres que l'utilisateur souhaite rentrer est conservé dans la variable `n`. Comme on ne sait pas, au moment où l'on écrit le programme, combien de nombres l'utilisateur va vouloir rentrer, on ne peut allouer de tableau de la bonne taille par une construction du genre `int t[5];` Au lieu de cela on a recours à la fonction `malloc` (*memory allocate*). L'interface de cette fonction est la suivante:

```
void *malloc(long size);
```

Cette fonction recherche un bloc de mémoire libre de taille égale à `size` octets, le marque occupé afin qu'aucune autre variable ne vienne l'utiliser et renvoie l'adresse de début de ce bloc sous forme d'un pointeur générique. Dans notre cas nous avons besoin d'un bloc pouvant contenir `n` `int`s pour stocker les `n` nombres que

va rentrer l'utilisateur. L'opérateur `sizeof(int)` retourne le nombre d'octets occupés par une variable de type `int`. Donc pour stocker `n` octets il faut `n*sizeof(int)` octets en mémoire.

Si la fonction `malloc` ne parvient pas à réserver un bloc de mémoire de la taille demandée, elle renvoie `NULL` (c'est-à-dire la valeur numérique zéro) en lieu et place de l'adresse de début de bloc. On teste donc si `ptr` vaut `NULL`, on signale une erreur dans ce cas et on interrompt le programme au moyen de la fonction `exit()`. Si l'appel à `malloc` se déroule correctement, l'adresse de début d'un bloc réservé est renvoyée et donc placée dans `ptr`. Pour fixer les idées supposons que l'utilisateur ait indiqué qu'il souhaitait rentre 5 nombres. A l'issue de l'appel à la fonction `malloc` la mémoire pourrait se retrouver dans l'état suivant:

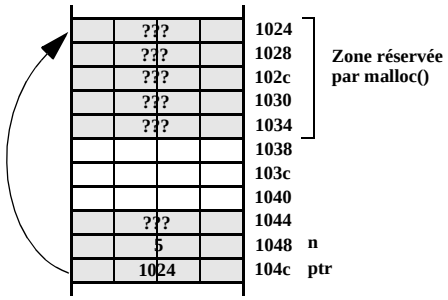


FIGURE 16: Organisation de la mémoire à l'issue du malloc (programme 62)

Dans cette figure on a grisé les zones de la mémoire utilisées ou réservées par notre programme. Les points d'interrogation indiquent que la valeur se trouvant dans la mémoire à cet endroit est indéterminée et peut être égale à n'importe quoi. A ce stade la fonction `malloc` a réservé de l'espace pour 5 `int`s à partir de l'adresse 1024 en mémoire. Cette adresse a été sauvée dans `ptr` qui est de type “pointeur sur `int`”.

Grâce à cette figure on s'aperçoit que l'on se trouve dans une situation tout à fait analogue à celle de la figure 13 où le pointeur `ip` permettait d'accéder aux éléments du tableau `t`. En fait on peut accéder à la zone de mémoire réservée par `malloc` comme à n'importe quel tableau d'entiers que nous avons rencontré jusqu'ici. Ainsi l'expression `ptr[0]` désigne l'entier contenu à l'adresse 1024, `ptr[1]` celui contenu à l'adresse 1028 et ainsi de suite jusqu'à l'indice `n-1`. Ainsi, à l'intérieur de la première boucle, l'expression `scanf(“%d”,&ptr[i]);` permet de stocker les valeurs entrées par l'utilisateur l'une après l'autre dans la zone réservée et l'expression `printf(“%d ”, ptr[i]);` de la deuxième boucle permet de les afficher.

Par équivalence de syntaxe on aurait également pu utiliser l'expression `ptr+i` au lieu de `&ptr[i]` et `*(ptr+i)` au lieu de `ptr[i]` (section 8.4).

La fonction `free`, quant à elle, libère une zone de mémoire allouée par la fonction `malloc`. Son interface est:

```
void free(void *ptr);
```

La mémoire utilisable par un programme est limitée, tout bloc alloué par `malloc` doit donc être libéré dès que possible afin de pouvoir être réutilisé plus tard par un nouvel appel à `malloc`. Un programme qui alloue répétitivement de la mémoire au moyen de `malloc` sans jamais la libérer avec `free` finit vite par épuiser la mémoire disponible. Dans cette situation, `malloc` ne peut plus faire de réservation et le programme a de fortes chances de planter. On voit que pour appeler `free`, il faut indiquer le début de la zone de mémoire à libérer au moyen d'un pointeur, il faut prendre garde à ne pas perdre la référence à des zones de mémoire. Dans l’exemple suivant, la zone réservée par le deuxième `malloc` ne pourra évidemment plus être retrouvée, car l’affectation à

`ptr2` l'efface. Le deuxième appel à `free` génère une erreur: `ptr1` et `ptr2` étant identiques, on tente de libérer

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

void main()
{
  int *ptr1, *ptr2;

  ptr1 = malloc(sizeof(int));
  ptr2 = malloc(sizeof(int));
  *ptr1 = 845;
  ptr2 = ptr1;
  printf("%d\n",*ptr2);
  free(ptr1);
  free(ptr2); /* !!! ERREUR !!! */
}
```

Programme 63

deux fois la même zone mémoire, ce qui n'est pas autorisé et peut, dans certains cas entraîner un plantage.

Les variables "normales" sont appelées *statiques* alors que celles qui sont pointées et allouées par `malloc` lors de l'exécution du programme sont appelées *dynamiques*. Le pointeur lui-même est donc en fait statique. Une variable dynamique est donc représentée par un symbole précédé de *.

9 Lecture ou écriture de fichiers sur le disque

Le langage C permet de sauvegarder des données dans un ou plusieurs fichiers sur disque après que l'exécution d'un programme soit terminée. On distingue *fichiers texte* et *fichiers binaires*.

Les fichiers texte sont les plus courants et leur avantage essentiel est qu'il est possible de les utiliser d'une machine à l'autre, même si les machines ont des microprocesseurs et des systèmes d'exploitation différents. Un autre avantage est qu'ils peuvent être lus et vérifiés par un observateur humain.

Les fichiers binaires permettent quant à eux de stocker des valeurs de variables sans devoir les transformer en chaînes de caractères. Toutefois on ne peut généralement pas les utiliser tels quels sur une autre plate-forme que celle sur laquelle ils ont été écrits, on doit pour cela prévoir des traitements tout à fait particuliers. De plus ils sont illisibles pour un observateur humain.

La manipulation de fichiers ne fait pas partie du langage C en lui-même mais utilise des fonctions qui sont toutefois normalisées et existent sur toutes les plates-formes informatiques où le C existe. Ces fonctions font partie de ce que l'on appelle la bibliothèque d'entrées/sorties standard. Il s'agit de la même bibliothèque, dite *stdio*, qui définit les fonctions `printf` et `scanf` que nous avons déjà utilisées. Les sections suivantes montrent qu'en fait ces deux fonctions ne sont que des cas particuliers d'autres fonctions, plus générales, permettant de manipuler des fichiers.

9.1 Fichiers texte

Les programmes qu'on a préparés, modifiés et exécutés jusqu'ici se trouvent sur le disque de l'ordinateur. Ce sont des fichiers. Il est évidemment possible de lire le contenu d'un fichier directement d'un programme et d'y déposer des valeurs pour les garder, même lorsqu'on éteint l'ordinateur.

Les instructions de lecture et d'écriture pour les fichiers sont nombreuses et seront résumées dans la section suivante. Nous verrons ici, simplement, les deux instructions de base permettant de lire ou d'écrire dans un fichier. Il s'agit de `fprintf` et de `fscanf`. Elles sont très semblables à `printf` et `scanf`.

```
/* écriture sur l'écran */
printf("Bonjour\n");

/* écriture sur le fichier lié à la variable monFichier */
fprintf(monFichier, "Bonjour\n");

/* lecture depuis le fichier lié à la variable monFichier */
fscanf(monFichier, "%d", &i);
```

Programme 64

On voit apparaître dans la ligne d'écriture sur fichier, un paramètre supplémentaire `monFichier`. Ce paramètre est ce que l'on appelle un *descripteur de fichier*. Il permet d'indiquer à la commande `fprintf` dans quel fichier elle doit écrire, en effet un programme peut tout à fait écrire dans plusieurs fichiers différents. La variable `monFichier` doit au préalable avoir eté déclarée:

```
FILE *monFichier;
```

puis le fichier doit être ouvert grâce à la commande `fopen` dont la syntaxe est:

```
monFichier = fopen("Nomdufichier.ext","mode");
```

où *`Nomdufichier.ext`* est le nom du fichier sur le disque et la chaîne *`mode`* indique ce qu'on veut faire avec le fichier. Les valeurs autorisées sont:

- `"r"` ouvre un fichier texte en lecture (on ne pourra pas écrire dedans)
- `"w"` ouvre un fichier texte en écriture. Si le fichier existait, son contenu est écrasé
- `"a"` ouvre un fichier texte en mode ajout. Si le fichier existait, il est ouvert et on se place à la fin de façon à pouvoir rajouter du texte dedans
- `"r+"` ouvre un fichier texte en mode mise à jour (lecture et écriture)
- `"w+"` ouvre un fichier texte en mode mise à jour. Si le fichier existait, son contenu est écrasé
- `"a+"` ouvre un fichier texte en mode mise à jour en se plaçant à la fin du fichier s'il existait déjà

La commande `fopen` renvoie un descripteur de fichier (type `FILE *`) si le fichier a été correctement ouvert ou bien `NULL` sinon.

Deux autres instructions sont très utilisées en C pour lire et écrire dans les fichiers texte, il s'agit de `fgets` et de `fputs` qui servent respectivement à lire une chaîne de caractère depuis un fichier et à écrire une chaîne de caractère dans un fichier. Leurs interfaces sont:

```
char *fgets(char *s, int n, FILE *file);
int fputs(char *s, FILE *file);
```

Nous avons déjà rencontré la fonction `fgets` pour lire des chaînes de caractères au clavier. Pour cela on utilisait `stdin` comme paramètre file. `stdin` est en fait un descripteur de fichier standard ouvert automatiquement par le système au début de l'exécution du programme (inutile donc d'utiliser `fopen` dans ce cas) et qui désigne le clavier. En effet pour Unix, le système d'exploitation sur lequel le langage C a été développé, tout ou à peu près tout est considéré comme un fichier, y compris le clavier et le terminal d'affichage. `fgets` lit jusqu'au caractère de fin de ligne (inclus) ou au maximum `n-1` caractères et les place dans `s` en ajoutant le zéro final. `fgets` renvoie un pointeur vers la chaîne lue (donc `s`) ou bien NULL si aucun caractère n'a pu être lu. `fscanf(file,"%s",s)` joue à peu près le même rôle si ce n'est que cette fonction s'arrête au premier espace rencontré s'il en apparaît un avant le caractère de fin de ligne.

`fputs` écrit la chaîne contenue dans le tableau de caractères `s` dans le fichier désigné par `file`.

Lorsque l'on a fini d'utiliser un fichier, il faut le fermer au moyen de la fonction `fclose`: `fclose(monFichier);`.

Lors d'une lecture, pour savoir si l'on est à la fin du fichier ou s'il y a encore des données à lire, on peut utiliser la fonction `feof(monFichier)`, ce qui signifie *end of file*. Cette fonction retourne une valeur non nulle (donc vraie d'un point de vue logique) lorsqu'on a lu tout le fichier. Le programme ci-dessous montre comment lire un fichier ligne par ligne, comment afficher chaque ligne sur l'écran, puis comment fermer le fichier:

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

void main()
{
  FILE *fichier;
  char nom_fichier[128], chaine[128];

  printf("Donnez le nom du fichier ? ");
  scanf("%127s",nom_fichier);

  if ((fichier = fopen(nom_fichier,"r")) == NULL) {
    printf("Erreur : Ouverture du fichier impossible\n");
    exit(1);
  }
  /* On a pu ouvrir le fichier */

  while (!feof(fichier)) {
    if (fgets(chaine, 128, fichier)!=NULL);
      printf("%s", chaine);
  }
}
```

Programme 65

Exercice 74. *Dans chaque ligne du fichier Dessin64.txt, on a déposé 4 nombres, correspondant aux coordonnées (x1 y1) et (x2 y2) des extrémités de vecteurs. On demande de lire ces nombres (fscanf(fichier,"%d %d %d %d",&x1,&y1,&x2,&y2)) et d'afficher sur l'écran graphique les vecteurs correspondants, partant de (x1, y1) et aboutissant à (x2, y2). Ces nombres sont compatibles avec l'échelle de l'écran graphique.*

Exercice 75. *Faire un programme qui copie un fichier sur un autre. Ce programme est pratiquement identique au programme 65. Il faut juste ouvrir un deuxième fichier pour y écrire dedans et fermer les deux fichiers à la fin. Faites ce programme avec fgets et fputs.*

Exercice 76. *Réaliser un programme qui compte le nombre de lignes et le nombre de caractères dans un fichier. Conseil: utiliser l'instruction strlen(*char).*

9.2 Résumé des fonctions de manipulation de fichiers

Cette section résume les fonctions C les plus courantes relatives à l'utilisation des fichiers:

```
FILE *entree ; /* fichier d'entrée */
FILE *sortie ; /* fichier de sortie */
x : integer ;
char chaine[128];

fichier = fopen("input.txt","r"); /* ouvrir en lecture le fichier input.txt */
fichier = fopen("output.txt","w"); /* ouvrir en écriture le fichier output.txt */
fscanf(entree,"%d\n",&x); /* lire dans entree la valeur de la variable x */
fgets(chaine,128,entree); /* lit une ligne de max. 128 caractères */
fgetc(char,entree); /* Lit un carctère dans le fichier entree */
fprintf (sortie, "Hello\n") ;
fprintf (sortie, "Bonjour %d\n",x) ; /* écrire dans le fichier sortie la chaîne de
                                        caractère "Bonjour" et la valeur de x */
fputs("Bonjour Monsieur",sortie); /*écrire dans sortie "Bonjour Monsieur" */
fputs(chaine,sortie); /* écrire dans sortie la valeur de la chaine chaine */
fputc(char,sortie); /* Met un caractère dans le fichier sortie */
fclose (entree) ;
fclose (sortie) ;
```

Programme 66

10 Structurer les données

Ce chapitre introduit le dernier concept fondamental du C, les *structures* ou *enregistrements*. Les structures permettent de grouper plusieurs variables et de leur donner un nom, au même titre qu'une fonction permet de grouper plusieurs instructions et de leur donner un nom. Les pointeurs permettent de créer des relations privilégiées entre variables et de créer des variables complexes dont la taille varie en cours d'exécution du programme. Par exemple, les pointeurs permettent de rajouter un élément au milieu d'un ensemble sans avoir à en recopier tous les éléments, ou créer une structure d'arbre (généalogique par exemple).

10.1 Types, l'instruction typedef

Nous avons rencontré jusqu'à présent un certain nombre de types de base offerts par le langage C: `int`, `float`, `double` qui sont des types numériques et `char` qui est le type caractère (pouvant être dans certaines circonstances considéré comme un type numérique également). Le langage C offre également une façon de définir ses propres types composés des types de base au moyen de l'instruction **`typedef`**:

```
typedef unsigned int uint;
typedef float VecteurT[4];
```

La première instruction fait du symbole `uint` un équivalent de `unsigned int`, la deuxième définit le symbole `VecteurT` comme étant un type représentant un tableau de 4 réels.

Ces nouveaux types peuvent ensuite être utilisés pour déclarer des variables exactement comme les types primitifs du langage:

```
uint i;
VecteurT x, y, z;
```

L'instruction `typedef` est utilisée tout particulièrement avec les structures présentées dans la section suivante.

10.2 Structures

Il arrive très souvent en programmation qu'une variable ne se décrive pas seulement par un entier, un réel ou une autre grandeur simple mais que sa description demande plusieurs informations de type différents. Par exemple, une variable décrivant une personne contient par exemple, son prénom et son nom (chaînes de caractère), sa date de naissance (3 entiers), le nom des parents, le numéro AVS, la taille, la couleur des yeux, etc...

Dans ce cas il devient nécessaire d'utiliser un mécanisme permettant de regrouper de façon cohérente un certain nombre de variables, c'est ce qu'offre le concept de structure. Dans les lignes suivantes on définit un type qui contient trois numéros et un nom.

```
typedef struct person
{
  char nom[32];
  char prenom[32];
  int jour, mois, annee;
  int index
} PersonT;
```

Programme 67

Cette structure, définit les champs `jour`, `mois`, `annee` de type `int`, les champs `nom`, `prenom` de type tableau de caractères, et le champ `index`, de type `int`. Il est tout à fait possible de définir des champs d'une structure qui soient eux-mêmes des structures et donc de placer des structures à l'intérieur de structures, à l'intérieur de structures, etc. La syntaxe employée ici permet de définir un nouveau type de variable appelé `PersonT` au moyen de l'instruction `typedef`. Grâce à cette définition de type on peut déclarer des variables du type structure avec la syntaxe suivante:

```
PersonT luke;
```

Sans cette définition de type il aurait fallu écrire:

```
struct person luke;
```

La définition du programme 67 peut se lire de la façon suivante: “une personne est caractérisée par son nom, son prénom, sa date de naissance, et un indice (par exemple, le numéro AVS)”.

Si l'on a déclaré une variable `luke` du type `PersonT` tel qu'il est défini ci-dessus, on peut désigner ses champs de la façon suivante en utilisant l'opérateur `.` :

```
PersonT luke;

luke.mois = 7;
strcpy(luke.nom,"Skywalker");
strcpy(luke.prenom,"Luke");
```

Programme 68

Attention, il est impératif de déclarer une variable du nouveau type avant toute utilisation, la définition du type en elle même ne permet pas d'affecter des valeurs.

Pour le champ `mois`, l'affectation est semblable à celle d'une variable normale. Pour le champ `nom` l'affectation est aussi semblable mais rappelons que pour une chaîne de caractères, on doit utiliser l'instruction `strcpy(tableau_char, chaine_a_copier);`. Les instructions du programme 68 peuvent se lire: le nom de la personne `luke` est ‘`Skywalker`’. Le `mois` de naissance de `luke` est le 7ème mois, etc...

Il est possible également de définir un tableau de personnes, ce qui est fait à la ligne suivante:

```
PersonT famille[4];
```

Dans ce cas, les champs des personnes se désignent de la façon suivante:

```
strcpy(famille[3].prenom, "Isabelle");
famille[2].annee = 1992;
```

Ces identificateurs sont formés d'une indication d'élément de tableau suivi de l'indication d'un champ: `.prenom`, `.mois`, etc... Ces lignes peuvent se lire: le prénom de la quatrième personne de la famille est `Isabelle`, l'année de naissance de la troisième personne de la famille est 1992.

Que gagne-t-on à utiliser des structures? Supposez que vous ayez un programme qui traite de trois personnes, Jules, Jacques et Jean. Sans structures, vous devriez déclarer 15 variables, et toutes les fonctions manipulant des personnes auraient 5 paramètres au minimum par personne manipulée (Prog. 69, droite). En utilisant les

structures, vous avez trois variables, et les fonctions manipulant les personnes ont un seul paramètre par personne (Prog. 69, gauche).

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

typedef struct person
{
  char nom[32];
  char prenom[32];
  int jour, mois, annee;
  int index;
} PersonT;

void PrintPerson(PersonT p)
{
  printf("%s %s\n",p.nom, p.prenom);
  printf("jour %d mois %d annee %d\n",
         p.jour, p.mois, p.annee);
}

void main()
{
  PersonT jules, jacques;
  PersonT jean = {"Tour", "Jean",
                  12, 4, 1958, 1};
  PrintPerson(jean);
}
```

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

typedef char StringT[32];

PrintPerson(char *nom, char *prenom, int jour,
            int mois, int annee)
{
  printf("%s %s\n",nom,prenom);
  printf("%d %d %d\n",jour, mois, annee);
}

void main()
{
  StringT nom_de_jules, prenom_de_jules;
  int jour_de_jules, mois_de_jules;
  int annee_de_jules;
  StringT nom_de_jacques, prenom_de_jacques;
  int jour_de_jacques, mois_de_jacques;
  int annee_de_jacques;
  StringT  nom_de_jean = “Tour”,
           prenom_de_jean = “Jean”;
  int  jour_de_jean=12,
       mois_de_jean = 4,
       annee_de_jean = 1958;

  PrintPerson(nom_de_jean,prenom_de_jean,
              jour_de_jean, mois_de_jean,
              annee_de_jean);
}
```

Programme 69

Il faut encore noter qu’on peut facilement affecter une structure entière à une autre comme on le fait avec des variables:

```
void main()
{
  PersonT jules, jacques, jean;
  ...
  jules = jean;
}
```

Programme 70

Par contre on ne peut pas comparer deux structures (il faut comparer champ par champ):

```
void main()
{
  PersonT jules, jacques, jean;
  ...
  if (jules == jean)...
}
```

Programme 71

Exercice 77. *La liste des personnes d'une famille est contenue dans le fichier* `famille.txt`. *Ecrire un programme avec une variable famille capable de stocker les informations d'au maximum 20 personnes, et copier les données du fichier dans cette variable.*

```
#define MAXSIZE 20

typedef struct PersonT
{
  ...
} PersonT;

void main()
{
  PersonT famille[MAXSIZE];
  ....
}
```

Programme 72

NB: la raison pour laquelle, dans le fichier famille.txt, on n'a pas mis toutes les informations concernant une personne sur la même ligne permet de mieux contrôler ce qu'on fait lors de la lecture.

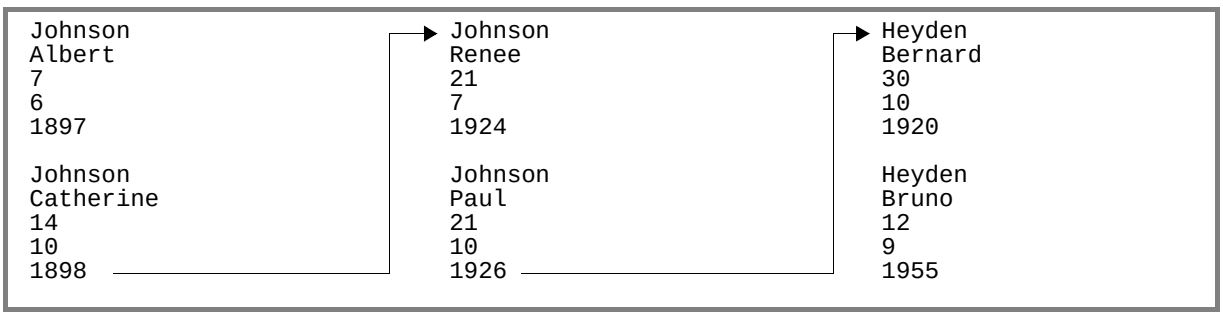


FIGURE 17: Contenu du fichier `famille.txt`

Exercice 78. *Triez les membres de la famille par ordre alphabétique (voir exercice 21).*

Exercice 79. *On se propose de modéliser un objet se déplaçant dans le plan (la fenêtre graphique) en décrivant sa position en x et en y (PosX, PosY), ainsi que sa vitesse en x et en y (SpeedX, SpeedY). Ces quatre coordonnées sont des nombres réels. La position est donnée en pixel, et la vitesse en pixel par itération du programme. A chaque itération du programme l'on ajoute la vitesse à la position, coordonnée par coordonnée, et l'on calcule la nouvelle vitesse de la façon suivante: Dist = sqrt (*$\mathrm{PosX}^2+\mathrm{PosY}^2$*),* Δ_{SpeedX}= -k.PosX/*(*Dist^3*),* Δ_{SpeedY}= -k.PosX/*(*Dist^3*). Exécutez 1000 itérations du programme, n utilisant comme valeurs initiales PosX = 150.0, PoxY = 0.0, SpeedX = 0.0, et SpeedY = 1.0. Utilisez les routines graphiques pour afficher l'objet à l'écran dans chacune de ses positions. La valeur à utiliser pour la constante k est 200. Le mouvement de l'objet vous rappelle-t-il quelque chose?*

Exercice 80. *Pour vous faciliter l'exercice suivant, plutôt que de déclarer 4 variables PoxX, PosY, SpeedX, SpeedY, déclarez une structure ObjectT avec 4 champs réels portant les noms PoxX, PosY, SpeedX, SpeedY, et une seule variable de type ObjectT. Ecrivez une procédure qui calcule la nouvelle position à partir de l'ancienne, et une procédure qui affiche la particule. Pensez aussi à recentrer l'origine lors de l'affichage. Le mouvement de l'objet est limité à l'intervalle (-200,200), tant en x qu'en y.*

Exercice 81. *Modifiez l'exercice 79 de façon à ce qu'il y ait trois objets, avec les valeurs initiales suivantes.*

```
Obj[1].PosX = 50.0 ; Obj[1].PoxY = 0.0 ; Obj[1].SpeedX = 0.0 ; Obj[1].SpeedY = 1.7 ;
Obj[2].PosX = 100.0 ; Obj[2].PoxY = 0.0 ; Obj[2].SpeedX = 0.0 ; Obj[2].SpeedY = 1.2 ;
Obj[3].PosX = 150.0 ; Obj[3].PoxY = 0.0 ; Obj[3].SpeedX = 0.0 ; Obj[3].SpeedY = 1.0 ;
```

Exercice 82. *Répéter l'exercice 74 (lecture et affichage de traits), mais en utilisant la structure suivante:*

```
#define MAXSIZE 200

typedef struct line
{
  int fromX, fromY ;
  int toX, toY;
} LineT;

LineT lines[MAXSIZE];
```

Programme 73

10.3 Structures, pointeurs de structures et fonctions

On peut tout à fait utiliser une structure comme argument ou valeur de retour d'une fonction. Dans ce cas, le passage s'effectue comme toujours par valeur, donc en copiant temporairement la structure, champ par champ. Ainsi dans le programme 74, lors de l'appel `Translate(p1, delta)`, les valeurs (3,5) et (1,1) de `p1` et `delta` sont copiées temporairement dans `p` et `deplacement`. Une fois le calcul effectué dans la variable tem-

```
#include <stdio.h>

typedef struct point
{
  int x;
  int y;
} PointT;

PointT Translate(PointT p, PointT deplacement)
{
  PointT temp;
  temp.x = p.x + deplacement.x;
  temp.y = p.y + deplacement.y;
  return temp;
}

main()
{
  PointT p1 = {3, 5};
  PointT delta = {1, 1};
  PointT p2;

  p2 = Translate(p1, delta);
  printf("%d %d\n",p2.x, p2.y);
}
```

Programme 74

poraire `temp`, sa valeur est retournée de la fonction et donc copiée dans `p2` grâce à l'affectation `p2=Translate(...)`.

Si les structures sont de très grande taille, le passage par valeur et donc la copie qu'il implique peut devenir particulièrement pénalisant. C'est pourquoi le passage de variables de type structure est plutôt rare, on lui pré-

fère généralement le passage par pointeur. Le programme suivant est similaire au précédent mais utilise des pointeurs:

```
#include <stdio.h>

typedef struct point
{
  int x;
  int y;
} PointT;

void Translate(PointT *p, PointT *deplacement)
{
  (*p).x += (*deplacement).x;
  (*p).y += (*deplacement).y;
}

main()
{
  PointT p1 = {3, 5};
  PointT delta = {1, 1};

  Translate(&p1, &delta);
  printf("%d %d\n",p1.x, p1.y);
}
```

Programme 75

La nouvelle fonction `Translate` ne renvoie plus le point translaté comme dans le programme 74 mais modifie directement, grâce au passage par pointeur, son premier argument.

Remarquez la notation utilisée pour accéder aux membres de la structure. Pour mémoire, si `ip` est un pointeur d'entier (`int *ip`), alors la notation `*ip` désigne l'entier pointé par `ip`, c'est-à-dire l'entier se trouvant à l'emplacement mémoire dont l'adresse est la valeur de `ip`. Il en va de même pour les structures, si `ptP` est un pointeur sur une structure de type point (`PointP *ptP`), alors la notation `*ptP` désigne la structure contenue dans l'emplacement mémoire commençant à l'adresse indiquée par `ptP`. On peut donc utiliser l'opérateur "`.`" pour accéder aux différents champs de cette structure, toutefois comme la priorité de cet opérateur est plus forte que celle de "`*`", on est obligé d'entourer `*ptP` de parenthèses: `(*ptP).x`. Mais les pointeurs de structure sont tellement fréquents que le C dispose d'un opérateur spécifique pour simplifier l'écriture d'une telle expression, l'opérateur "`->`". Cet opérateur nécessite un pointeur de structure à sa gauche et un identificateur de champ à sa droite. Ainsi l'expression `(*ptP).x` est en tout point identique à `ptP->x`. Un programmeur C exercé écrirait donc la fonction `Translate` du programme 75, comme suit:

```
void Translate(PointT *p, PointT *deplacement)
{
  p->x += deplacement->x;
  p->y += deplacement->y;
}
```

10.4 Utilisation des pointeurs pour améliorer les programmes

Nous allons reprendre l'exercice 78 (tri de la famille) et l'améliorer de deux façons différentes: limiter la taille mémoire utilisée, et accélérer le tri.

10.5 Réduction de la mémoire occupée par un programme

La première difficulté liée à l'exercice 77 est son usage excessif de mémoire. Si l'on considère une valeur `MAX_SIZE=10000`, et la taille de la structure `PersonT` = 100 octets, la taille mémoire requise pour stocker la variable `famille` est 10^6 octets. Ceci même si le fichier `famille.txt` ne contient qu'une seule personne.

On peut améliorer la situation en déclarant non plus un tableau de personnes, mais un tableau de pointeurs vers des personnes, de la façon suivante[1]:

```
typedef struct personT
{
...
} PersonT;

void main()
{
  PersonT *famille[MAXSIZE];
  ...
}
```

Programme 76

La taille d'un pointeur est en général 4 octets (pour les machines dites 32 bits, et 8 octets pour les machines dites 64 bits). De nouveau, si `MAXSIZE=10000`, la taille occupée par le programme en mémoire est 4.10^4 + 100*NbrePersonneDsFichier. Dans le cas où le fichier `famille.txt` contient six personnes, le programme 76 occupe 40600 octets contre 1040000 octets pour le programme 72. C'est un facteur 25 d'économie.

Exercice 83. *Réécrivez l'exercice 77 (lecture du fichier famille.txt) en utilisant les déclarations du programme* 76.

La figure 18 montre graphiquement l'utilisation de la mémoire dans les deux situations:

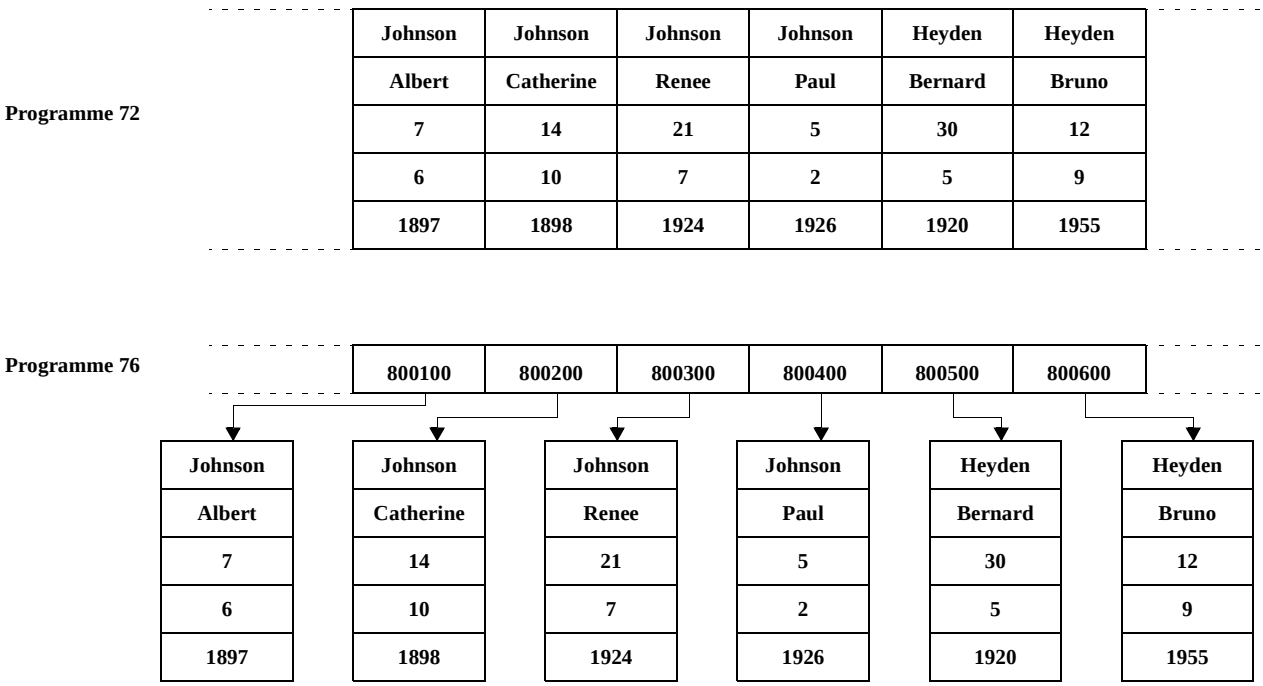


FIGURE 18: Deux organisations mémoire pour les programmes 72 et 76

Exercice 84. *Réécrivez l'exercice 78 (tri des membres de la famille) en utilisant les déclarations du programme* 76.

1. On pourrait aussi utiliser les tableaux dynamiques mais ils amènent ensuite les mêmes problèmes que les tableaux statiques lorsqu'il s'agit de déplacer les données bien qu'ils permettent un gain de place en mémoire.

Exercice 85. *Réécrivez l'exercice 82 (lire les points dans un fichier) en utilisant les déclarations suivantes (tableau de pointeurs de ligne, plutôt que tableau de lignes):*

```
#define MAXSIZE 20
typedef struct lineT
{
  int fromX, fromY;
  int toX, toY;
} LineT;

void main()
{
  LineT *Lines[MAXSIZE];
  ...
}
```

Programme 77

10.6 Amélioration de la performance de la routine de tri

Jusqu'à présent, pour trier les personnes organisées en tableau (voir exercice 78), à chaque itération de l'algorithme de tri, on échange tout le contenu de deux structures `PersonT` chaque fois qu'ils sont dans le désordre (programme 78). Pour référence, la fonction `Older` retourne un booléen qui indique si la première personne est plus âgée que la seconde. Il s'agit donc d'un tri sur l'âge, et non sur le nom. Chaque échange demande trois copies. Par exemple, dans le pire des cas, le tri de 1000 personnes demande 999*998/2 (soit environ 500.000) échanges, soit le mouvement d'environ 5.10^5*3*100 octets (100 octets est la taille d'une structure), soit près de 150 Moctets.

```
/* Tri1 */
#define SIZE ...
typedef struct person {...} PersonT;

void main()
{
  PersonT famille[SIZE];
  PersonT tmp;
  ...
  for(i = 0; i < SIZE-1; i++)
    for (j = i+1; j < SIZE; j++)
      if (Older(famille[i], famille[j]))
        {
          tmp = famille[i];
          famille[i] = famille[j];
          famille[j] = tmp;
        }
}
```

Programme 78

Grâce aux pointeurs, on peut améliorer sensiblement ce programme. Dans le programme 79, la variable `famille` est déclarée comme tableau de pointeurs, et dans la routine de tri, on échange non plus des structures,

mais des pointeurs. Au bout du compte, le résultat est le même, mais on a copié 25 fois moins de données (une structure `PersonT` représente 100 octets, tandis qu'un pointeur n'en représente que 4).

```
/* Tri2 */
#define SIZE ...
typedef struct person {...} PersonT;

void main()
{
  PersonT *famille[SIZE];
  PersonT *tmp;
  ...
  for(i = 0; i < SIZE-1; i++)
    for (j = i+1; j < SIZE; j++)
      if (Older(famille[i], famille[j]))
        {
          tmp = famille[i];
          famille[i] = famille[j];
          famille[j] = tmp;
        }
}
```

Programme 79

Observez le programme 80. Pouvez-vous dire combien d'octets sont copiés dans le pire des cas de tri d'une famille de 1000 personnes? Ce programme illustre la différence qu'il y a entre la copie de pointeurs (programme 79, `famille[i] = famille[j]`) et la copie de structures (programme 80, `*famille[i] = *famille[j]`).

```
/* Tri3 */
#define SIZE ...
typedef struct person {...} PersonT;

void main()
{
  PersonT *famille[SIZE];
  PersonT *tmp;
  ...
  for(i = 0; i < SIZE-1; i++)
    for (j = i+1; j < SIZE; j++)
      if (Older(famille[i], famille[j]))
        {
          *tmp = *famille[i];
          *famille[i] = *famille[j];
          *famille[j] = *tmp;
        }
}
```

Programme 80

La figure 19 illustre l'organisation de la mémoire après les programmes 79 (Tri2) et 80 (Tri3). L'organisation initiale de la mémoire est celle de la figure 18 (bas). Après l'exécution du programme Tri2, seuls les pointeurs

ont changé. Après le programme Tri3, les pointeurs n'ont pas changé, mais les données dans les structures, elles, ont changé.

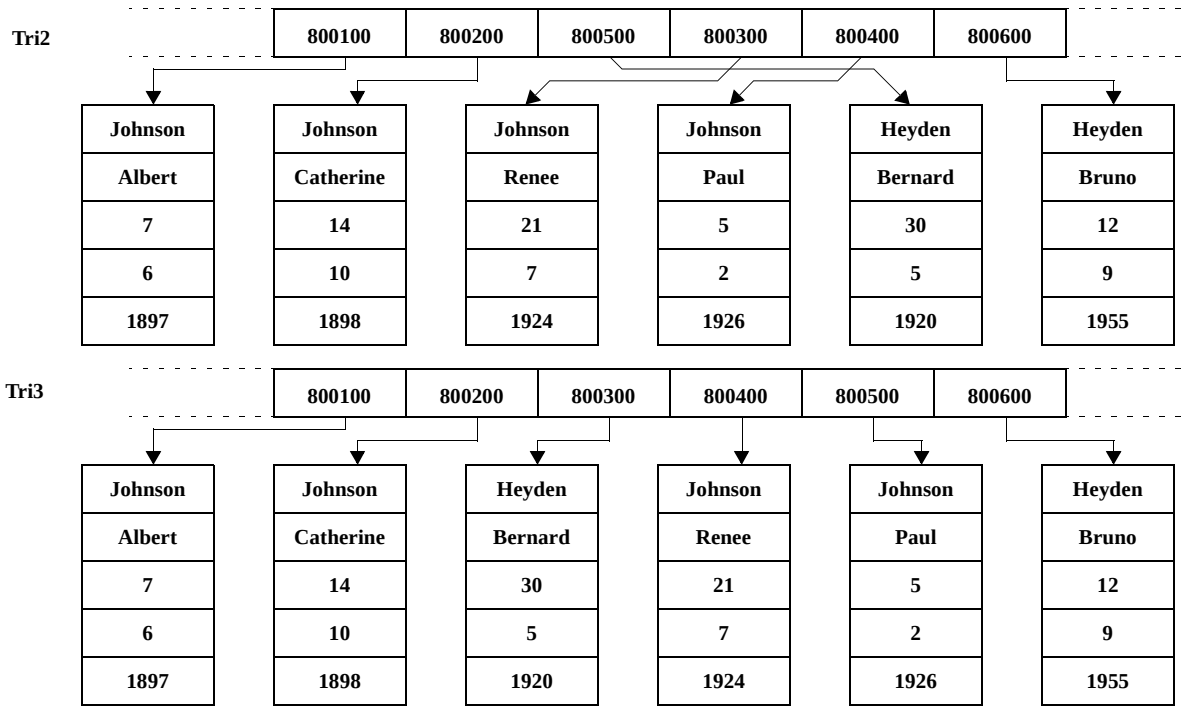


FIGURE 19: Organisation mémoire après les programmes Tri2 et Tri3

10.7 Utilisation des pointeurs pour faire un 'arbre généalogique'

Dans les schémas, nous avons souvent représenté jusqu'ici les pointeurs sous forme de flèches. Que ce soit cette représentation ou le nom même de pointeur, tout indique qu'un pointeur permet d'établir une relation entre deux emplacements de la mémoire, c'est-à-dire entre deux variables. Imaginons que l'on souhaite créer une relation de parenté entre les personnes d'une famille (voir exercice 77), c'est-à-dire pouvoir être capable de dire pour chaque personne, quel est son père et quelle est sa mère. Comme toutes les personnes sont caractérisées par un index unique, on pourrait établir cette relation au moyen de deux variables de type `int` contenues dans la structure et qui contiendraient l'index du père et l'index de la mère. Cette méthode a une limitation toutefois: il n'est facile de retrouver les informations relatives au père et à la mère que tant que l'on peut accéder facilement à un enregistrement à partir de son numéro d'index. C'était le cas dans les exemple précédents puisque l'index d'une personne correspondait à un indice dans le tableau des structures.

Imaginons maintenant que les structures définissant les personnes soient dispersées et que leur index ne permette pas de retrouver facilement où elles sont rangées en mémoire. Comment maintenir alors une relation d'une personne vers ses parents? Une solution passe par l'utilisation de pointeurs. En effet chaque structure contenant les données relatives à une personne est présente à un endroit bien déterminé de la mémoire et, de ce fait, se trouve identifiée de façon unique par son adresse. Ainsi plutôt que d'associer à chaque personne les indexes de son père et de sa mère, on peut lui associer deux pointeurs contenant l'adresse en mémoire où l'on

peut trouver les structures décrivant son père et sa mère. Pour cela, on rajoute donc dans la structure `PersonT` deux champs: `pere` et `mere` de type `struct person *variable`[1].

```
typedef struct person
{
  char nom[32];
  char prenom[32];
  int jour, mois, annee;
  int index;
  struct person *pere;
  struct person *mere;
} PersonT;
```

Programme 81

Dans le fichier `famille2.txt`, dont le contenu est montré dans la Figure 20, les relations de parenté sont indiquées: Renee et Paul ont pour parents Albert et Catherine. Bruno a pour parents Bernard et Renee.

```
Johnson
Albert
7
6
1897
inconnu
inconnu
inconnu
inconnu

Johnson
Catherine
14
10
1898
inconnu
inconnu
inconnu
inconnu
```

```
Johnson
Renee
21
7
1924
Johnson
Albert
Johnson
Catherine

Johnson
Paul
21
10
1926
Johnson
Albert
Johnson
Catherine
```

```
Heyden
Bernard
30
10
1920
inconnu
inconnu
inconnu
inconnu

Heyden
Bruno
12
9
1955
Heyden
Bernard
Johnson
Renee
```

FIGURE 20: Contenu du fichier famille2.txt

Exercice 86. *On vous donne un fichier* `famille2.txt`*, qui contient pour chacun des membres de la famille le nom du père et de la mère s'ils sont connus, 'inconnu' autrement. Modifiez le programme 77 de façon à ce qu'il lise les informations contenues dans le fichier* `famille2.txt` *et initialise tous les champs de la structure* `PersonT`*. Si le père ou la mère sont inconnus, les champs* `pere` *et* `mere` *doivent être initialisés à* `NULL`*. Utilisez la fonction* `strcmp(chaîne1, chaîne2)` *qui renvoie 0 si les deux chaînes sont identiques. Cette fonction est déclarée dans le fichier* `string.h`

1. L'identificateur `PersonT` n'est utilisable qu'après le point-virgule marquant la fin de l'instruction `typedef`. A l'intérieur de la définition de la structure, on doit donc utiliser `struct person`.

Exercice 87. *Reprenez l'exercice 82 (lire et afficher des lignes), avec la structure suivante.*

```
#define MAXSIZE 20
typedef struct line
{
  int fromX, fromY;
  int toX, toY;
  int index;
  struct line *nextP;
} LineT;

LineT *lines[MAXSIZE];
```

Programme 82

Exercice 88. *Après avoir lu tous les points, initialisez le champ nextP de chaque ligne* `l` *en trouvant la ligne* `m` *tel que* `l.toX` *et* `l.toY` *soient égaux à* `m.fromX` *et* `m.fromY` *respectivement. Arrangez-vous (en utilisant le champ* `nextP` *maintenant initialisé) pour que la boucle affiche les traits consécutifs l'un après l'autre (et non dans l'ordre dans lequel ils sont dans le fichier). Rajoutez un délai dans la boucle d'affichage, et vérifiez visuellement que les traits sont affichés dans l'ordre. Utilisez par exemple le fichier Dessin64c.txt. Pour se simplifier un peu la vie, on ajoute comme information qu'il s'agit d'une seule ligne continue (et non de plusieurs segments).*

Exercice 89. *Revenons à l'exercice 86. Plutôt que de "matérialiser" les relations père/mère, on souhaite matérialiser la relation enfants, en ajoutant dans chaque structure un tableau de pointeurs vers des personnes, de la façon suivante:*

```
typedef struct person
{
  char nom[32];
  char prenom[32];
  int jour, mois, annee;
  int index;
  struct person *enfants[4];
} PersonT;
```

Programme 83

Ecrivez un programme qui lit le fichier `famille2.txt`*, et initialise tous ses champs de façon à pouvoir retrouver directement tous les enfants de chaque personne.*

10.8 Utilisation des pointeurs dans les listes chaînées

Il subsiste encore trois difficultés dans la solution proposée à l'exercice 83:

- on reste limité dans la taille de l'arbre généalogique que le programme peut traiter car le tableau stockant les structures de personnes ou les pointeurs vers ces structures est de taille limitée fixée au moment où l'on écrit le programme;
- il est difficile de rajouter une personne au milieu du tableau de personnes (pour faire cela, il faut décaler toutes les personnes qui "suivent", ce qui occasionne pas mal de copies de structures ou de pointeurs).

Dans la section précédente, nous avons vu comment les pointeurs permettent de mettre en relation des structures représentant des personnes. La relation choisie pour illustrer cette possibilité était une relation de parenté (une personne étant donnée, les pointeurs `pere` et `mere` permettaient d'identifier ses parents). Les pointeurs peuvent également apporter une solution aux difficultés évoquées plus haut. La solution lorsque l'on souhaite pouvoir traiter un nombre illimité de structures est de ne déclarer dans le programme qu'un seul pointeur vers la première structure et dans la définition de la structure elle-même, de prévoir un pointeur qui permettra de pointer vers la structure suivante. on établit ainsi une relation similaire à la relation de parenté de la

section 10.7 mais il s'agit dans ce cas d'une simple relation d'ordre: la première personne pointe sur la deuxième, qui elle-même pointe vers la troisième, qui elle-même pointe vers la quatrième et ainsi de suite jusqu'à la dernière qui, elle, ne pointe sur rien (pointeur `NULL`). De cette façon, on peut toujours rajouter une structure supplémentaire, la seule limite étant celle de la mémoire disponible et non pas une limite arbitraire fixée lors de l'écriture du programme. On peut généraliser cette méthode en déclarant deux pointeurs dans chaque structure: l'un vers la structure suivante, l'autre vers la précédente.

Les structures se retrouvent ainsi organisées selon ce qu'on appelle une *liste chaînée simple* (dans le cas où chaque structure n'a qu'un pointeur vers la suivante) ou *doublement chaînée* (dans le cas où chaque structure pointe vers la précédente et la suivante).

Pour illustrer cela, voici un programme (programme 84) qui lit le fichier `famille.txt`, et insère chacun des membres dans une telle liste chaînée:

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

typedef struct person
{
  char nom[32], prenom[32];
  int jour, mois, annee, index;
  struct person *next;
} PersonT;

main()
{
  PersonT *teteDeListe, *person;
  FILE *fichier;
  int i;
  char temp[5];

  teteDeListe = NULL;
  if ((fichier = fopen("famille.txt","r")) == NULL) {
    printf("Erreur \n");
    exit(1);
  }

  i = 1;
  while (!feof(fichier)) {
    person = (PersonT *)malloc(sizeof(PersonT));
    person->index = i;
    fgets(person->nom, 32, fichier);
    fgets(person->prenom, 32, fichier);
    fscanf(fichier,"%d", &person->jour);
    fscanf(fichier,"%d", &person->mois);
    fscanf(fichier,"%d", &person->annee);
    fgets(temp, 5, fichier);
    person->next = teteDeListe;
    teteDeListe = person;
    i++;
  }

  /* Affichage des données lues */
  person = teteDeListe;
  while (person != NULL) {
    printf("%s",person->prenom);
    printf("%s\n",person->nom);
    person = person->next;
  }
}
```

Programme 84

Dans ce programme, chaque personne (représentée par la structure `PersonT`) est définie de façon classique, par son nom, son prénom, sa date de naissance et un indice. On remarque également le champ supplémentaire, `next`, qui permet donc d'établir une relation entre une personne et la personne suivante dans la liste. La famille est organisée de la façon suivante: la première personne de la famille est en relation avec la deuxième, qui est elle-même en relation avec la troisième, qui est elle-même en relation avec la quatrième, etc... La variable

`teteDeListe` indique juste où se trouve la première personne de la famille, à partir de cette personne on peut accéder à la deuxième, de la deuxième on peut accéder à la troisième et ainsi de suite de proche en proche pour n'importe quelle personne de la famille. On a ainsi bien créé une liste de personnes que l'on peut entièrement parcourir en commençant par la personne pointée par `teteDeListe`.

Le programme 84 lit dans le fichier `famille.txt` les personnes une par une (première boucle `while`). On alloue au fur et à mesure l'espace mémoire nécessaire pour contenir les informations relatives à chacune des personnes (`person = malloc(sizeof(PersonT))`), initialise les informations concernant la personne à partir des données du fichiers (les instructions `fgets` et `fscanf` dans la boucle), et insère la personne dans la liste chaînée dont l'adresse de début est contenue dans `teteDeListe`. Etudions les deux instructions d'insertion dans la liste chaînée:

```
person->next = tetedeListe ;
teteDeListe = person ;
```

Pour des questions d'efficacité, chaque nouvelle personne est insérée au début de la liste. Pour la nouvelle personne ajoutée, la personne suivante est celle se trouvant actuellement en tête de liste. C'est le sens de la première affectation:

```
person->next = tetedeListe;
```

Afin de mieux comprendre, on peut représenter graphiquement, cette opération:

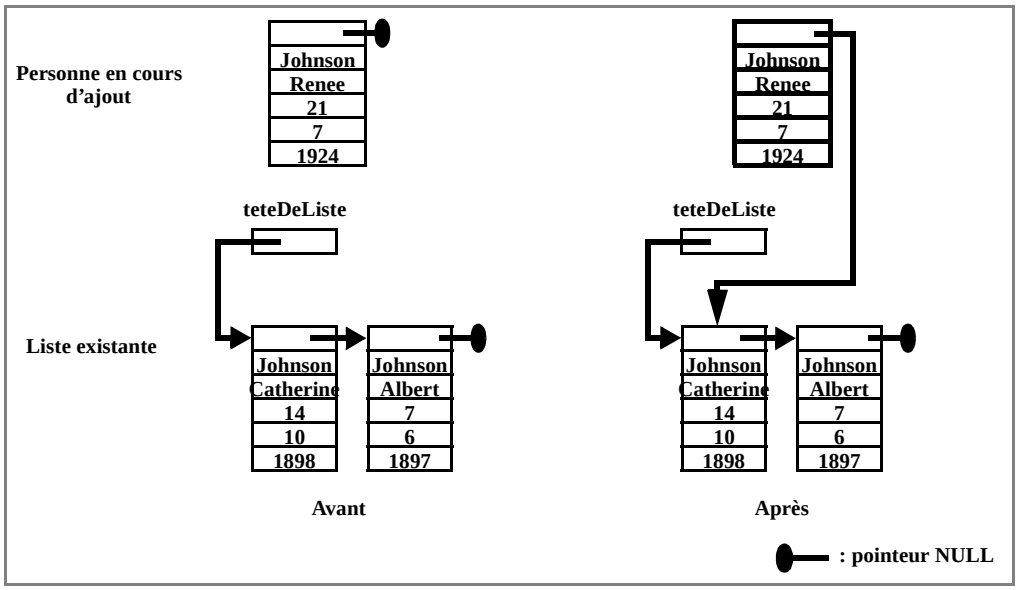


FIGURE 21: Effet de l'affectation `person->next = tetedeListe;`

A ce stade, la personne en cours d'ajout (Renée Johnson) est en quelque sorte déjà insérée dans la liste puisqu'à partir d'elle on peut atteindre la personne suivante (Catherine Johnson) et à partir de là toutes les autres

personnes déjà présentes dans la liste. Renée Johnson est de fait devenue la nouvelle tête de liste. La deuxième affectation met alors à jour le pointeur `teteDeListe` afin qu'il pointe vers cette nouvelle tête de liste:

```
teteDeListe = person;
```

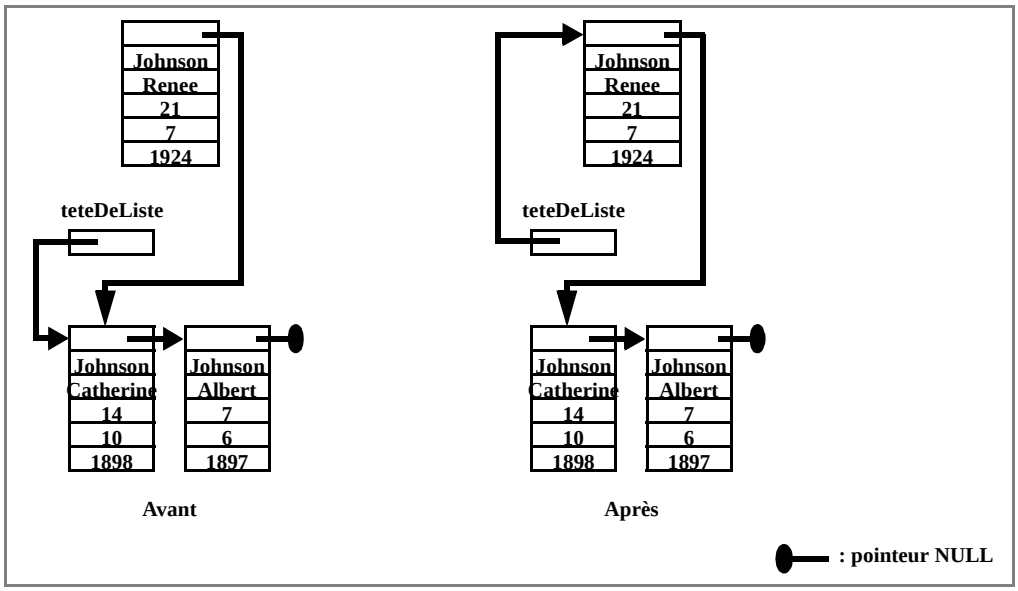


FIGURE 22: Effet de l'affectation `teteDeListe = person;`

Pour résumer, la figure 23 représente l'ajout des trois premières personnes dans la liste chaînée (ainsi que son état final).

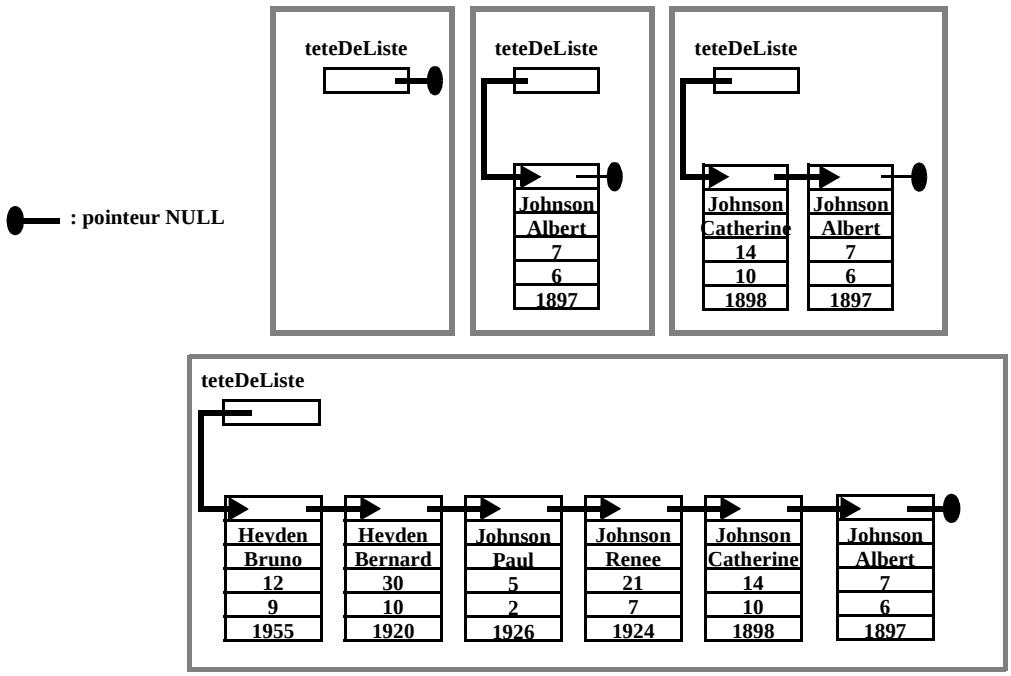


FIGURE 23: Les différents états de la liste chaînée

La dernière boucle du programme affiche toutes les personnes dans la liste. Remarquez que les personnes sont dans la liste en ordre inverse par rapport au fichier. Il est bien entendu possible de mettre les personnes dans la liste dans le même ordre que dans le fichier (voir exercice 90). Cela peut cependant être beaucoup

moins efficace. En effet, pour mettre une personne au bout de la liste, il faut parcourir toute la liste à chaque insertion. Par exemple pour insérer la dixième personne dans la liste, il faut faire une boucle qui passe en revue les neuf premières personnes de la liste, et ajoute la dixième personne au bout de la liste. En général pour créer une liste de n personnes en les insérant une par une en bout de liste, il faut passer en revue: $\sum_{i=0}^{n-1} i = (n(n-1))/2$ personnes, soit de l'ordre de n^2 personnes, ce qui devient très lent lorsque le nombre de personnes est grand.

Pourquoi utiliser des listes, alors que les tableaux fonctionnaient à peu près? Avec les listes chaînées, il n'y a pas moyen d'accéder directement aux personnes de la famille en utilisant les indices du tableau. Par contre, on peut agrandir "sans limite" le nombre de personnes dans la famille. Tout dépend de la situation dans laquelle on se trouve. Il y a beaucoup de situations où l'on ne peut pas se permettre d'allouer un tableau de taille fixe qui soit suffisamment grand pour traiter le pire des cas. A ce moment-là, la seule solution est la liste chaînée.

Exercice 90. *Modifier le programme* 84 *(programme ~gennart/exercices-corriges/program*84.*c) sans ajouter de variables de façon à ce qu'il ajoute les personnes au bout de la liste chaînée. Montrer que ce programme est beaucoup plus lent que le programme* 84 *(par exemple en insérant 1000, 2000 et 4000 personnes dans la liste).*

Exercice 91. *Modifier l'exercice précédent en ajoutant une variable* `PersonT *finDeListe` *qui pointe toujours vers la dernière personne de la liste. Ceci permet d'insérer une nouvelle personne à la fin de la liste chaînée de façon beaucoup plus rapide.*

Le programme 85 montre un programme qui traite une liste simplement chaînée de personnes et les trie par nom de famille et prénom. Le principe général de la routine de tri est des plus simples: on cherche à construire une nouvelle liste triée à partir de la liste originale. Pour cela, on parcourt la liste originale et on trouve son plus grand élément (lexicographiquement parlant), noté `max`. Cet élément est enlevé de la liste originale et rajouté dans la nouvelle liste. On recommence ce processus jusqu'à ce que la liste originale soit vide. Il faut noter que pour enlever un élément d'une liste simplement chaînée il faut disposer d'un pointeur sur l'élément précédent car il faut rattacher cet élément à celui qui suit celui qu'on enlève (figure 24), c'est pour cela que l'on utilise les

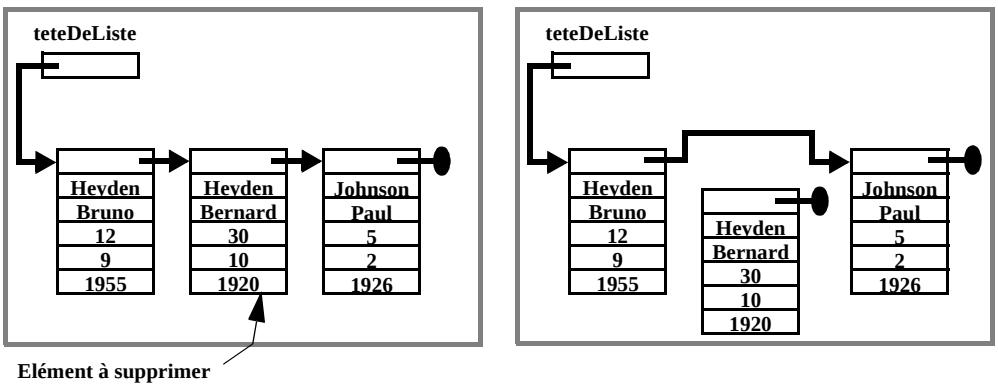


FIGURE 24: Suppression d'un élément d'une liste simplement chaînée

variables `prev` et `maxPrev`. Par contre lorsque l'élément à enlever est en tête de liste, il suffit de changer le pointeur de tête de liste pour le faire pointer vers l'élément suivant.

Le programme 86 montre un programme qui traite une liste circulaire doublement chaînée de personnes. Une liste circulaire est une liste où l'élément suivant du dernier est le premier (figure 25). Un algorithme de parcours d'une telle liste qui se contenterait de passer d'un élément au suivant sans autre forme de précaution n'aurait pas de fin. Une astuce pour éviter ce genre de problème ainsi que d'avoir à traiter l'insertion du premier élément et la suppression du dernier comme des cas particuliers, consiste à faire en sorte que la liste contienne toujours un élément initial vide, appelé sentinelle.

Nous laissons au lecteur le soin de s'armer d'un crayon, d'un papier et d'un peu de patience afin de faire les schémas nécessaires pour étudier et comprendre ces programmes.

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

typedef struct person
{
  char nom[32], prenom[32];
  int jour, mois, annee;
  struct person *next;
} PersonT;

PersonT *teteDeListe;

void LitAdresses(char *nom_fichier)
{
  FILE *fichier;
  PersonT *person;
  char temp[5];

  teteDeListe = NULL;

  if ((fichier = fopen(nom_fichier,"r"))
       == NULL) {
    printf("Erreur \n");
    exit(1);
  }
  while (!feof(fichier)) {
    person = malloc(sizeof(PersonT));
    fgets(person->nom, 32, fichier);
    fgets(person->prenom, 32, fichier);
    fscanf(fichier,"%d", &person->jour);
    fscanf(fichier,"%d", &person->mois);
    fscanf(fichier,"%d", &person->annee);
    /* absorbe le retour chariot
       sur la ligne courante */
    fgets(temp,5,fichier);
    /* absorbe la ligen vide */
    fgets(temp,5,fichier);
    person->next = teteDeListe;
    teteDeListe = person;
  }
  fclose(fichier);
}

void AfficheAdresses()
{
  PersonT *person;
  for (person=teteDeListe;
       person != NULL;
       person=person->next) {
    printf("%s",person->nom);
    printf("%s",person->prenom);
    printf("%d\n",person->jour);
    printf("%d\n",person->mois);
    printf("%d\n",person->annee);
    printf("\n");
  }
}
```

```
void TrieAdresses()
{
  PersonT  *nouvelleTete, *person, *prev,
           *max, *maxPrev;
  int i;

  nouvelleTete = NULL;

  while (teteDeListe) {
    maxPrev = NULL;
    prev = max = teteDeListe;
    printf("Step %d\n", i);
    AfficheAdresses();
    for (person=teteDeListe->next;
         person!=NULL;
         person=person->next) {
      if ((strcmp(person->nom,
                  max->nom) > 0) ||
          (strcmp(person->nom,
                  max->nom)==0 &&
           strcmp(person->prenom,
                  max->prenom) > 0)) {
        maxPrev = prev;
        max = person;
      }
      prev = person;
    }
    /* Si le max n'etait pas en tete de
       liste, on l'enleve de la liste en
       raccrochant son precendent a son
       suivant. Sinon on l'enleve en
       faisant avancer la tete de liste
       d'une personne */
    if (maxPrev)
      maxPrev->next = max->next;
    else
      teteDeListe = teteDeListe->next;
    /* On insere le max en tete de la
       nouvelle liste */
    max->next = nouvelleTete;
    nouvelleTete = max;
    i++;
  }
  teteDeListe = nouvelleTete;
}

void main()
{
    LitAdresses("famille.txt");
    TrieAdresses();
    printf("Resultat\n");
    AfficheAdresses();
}
```

Programme 85

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

typedef struct person
{
  char nom[32], prenom[32];
  int jour, mois, annee;
  struct person *next, *prev;
} PersonT;

PersonT *teteDeListe;

PersonT *CreeListe()
{
  PersonT *nouvelleListe;

  nouvelleListe = malloc(sizeof(PersonT));
  nouvelleListe->next = nouvelleListe;
  nouvelleListe->prev = nouvelleListe;
  return nouvelleListe;
}

void AjoutePersonne(PersonT *listPerson,
                    PersonT *newPerson)
{
  newPerson->next = listPerson;
  newPerson->prev = listPerson->prev;
  listPerson->prev->next = newPerson;
  listPerson->prev = newPerson;
}

void EnlevePersonne(PersonT *person)
{
  person->prev->next = person->next;
  person->next->prev = person->prev;
}

void LitAdresses(char *nom_fichier)
{
  FILE *fichier;
  PersonT *person;
  char temp[5];

  teteDeListe = CreeListe();

  if ((fichier = fopen(nom_fichier,"r"))
       == NULL) {
    printf("Erreur \n");
    exit(1);
  }
  while (!feof(fichier)) {
    person = malloc(sizeof(PersonT));
    fgets(person->nom, 32, fichier);
    fgets(person->prenom, 32, fichier);
    fscanf(fichier,"%d", &person->jour);
    fscanf(fichier,"%d", &person->mois);
    fscanf(fichier,"%d", &person->annee);
    /* absorbe le retour chariot
       sur la ligne courante */
    fgets(temp,5,fichier);
    /* absorbe la ligne vide */
    fgets(temp,5,fichier);
    AjoutePersonne(teteDeListe, person);
  }
  fclose(fichier);
}
```

```
void AfficheAdresses()
{
  PersonT *person;
  for (person = teteDeListe->next;
       person != teteDeListe;
       person = person->next) {
    printf("%s",person->nom);
    printf("%s",person->prenom);
    printf("%d\n",person->jour);
    printf("%d\n",person->mois);
    printf("%d\n",person->annee);
    printf("\n");
  }
}

void TrieAdresses()
{
  PersonT *nouvelleTete, *person, *prev,
          *min, *maxPrev;
  int i;

  nouvelleTete = CreeListe();

  while (teteDeListe->next!=teteDeListe) {
    printf("Step %d\n", i);
    AfficheAdresses();
    min = teteDeListe->next;
    for (person=min->next;
         person!=teteDeListe;
         person=person->next) {
      if ((strcmp(person->nom,
                  min->nom) < 0) ||
          (strcmp(person->nom,
                  min->nom)==0 &&
            strcmp(person->prenom,
                   min->prenom) < 0)) {
        min = person;
      }
    }
    EnlevePersonne(min);
    AjoutePersonne(nouvelleTete, min);
    i++;
  }
  free(teteDeListe);
  teteDeListe = nouvelleTete;
}


main()
{
  LitAdresses("famille.txt");
  TrieAdresses();
  printf("Resultat\n");
  AfficheAdresses();
}
```

Programme 86

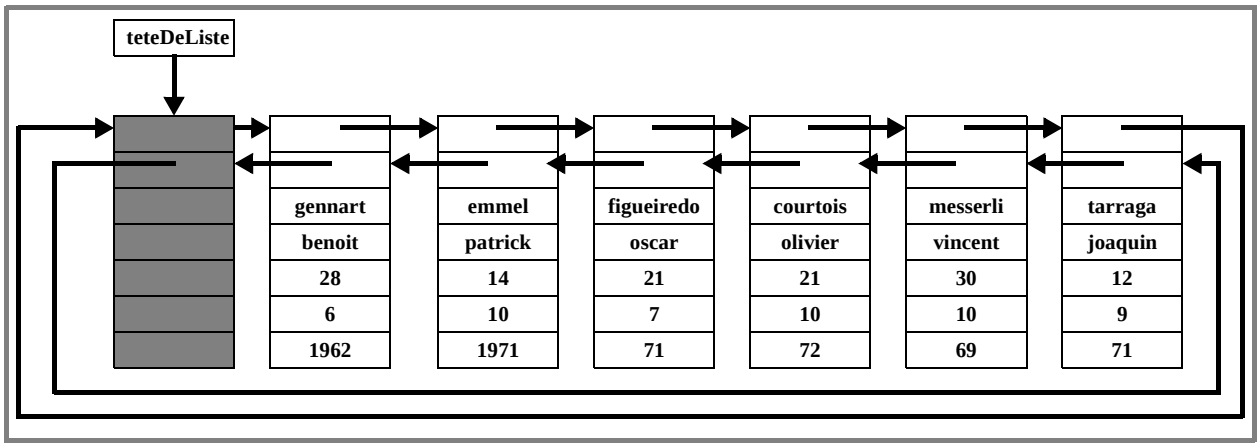


FIGURE 25: Liste circulaire doublement chaînée

Exercice 92. *Modifier le programme 86 pour que la routine qui lise les noms introduise immédiatement les personnes dans l'ordre alphabétique, au fur et à mesure que les noms sont lus dans le fichier.*

Exercice 93. *Arbre. A préparer.*

Exercice 94. *Table associative. A préparer.*

Exercice 95. *Graphe acyclique dirigé. A préparer.*

10.9 Discussion de ce chapitre

Dans ce chapitre, vous avons envisagé trois façons de résoudre le même problème: tableau de structures, tableau de pointeurs vers des structures, listes. Quelle solution faut-il choisir ? Cela dépend du problème.

Les tableaux de structures sont faciles à utiliser, et l'accès à chacun des éléments est immédiat. La taille maximale est fixe ou variable. On ne peut cependant pas rajouter d'éléments au milieu du tableau sans copie de structures, et le mouvement d'objet demande des copies.

Les tableaux de pointeurs sont un peu plus délicats à utiliser. L'accès à chacun des éléments reste immédiat. La taille maximale est fixe ou variable. Rajouter un élément au milieu du tableau ou en déplacer un demande seulement des copies de pointeurs.

Les listes demandent pas mal de travail en C. On perd l'accès immédiat aux éléments de la liste (il faut suivre la liste pour trouver l'élément que l'on cherche). L'avantage est que la longueur de la liste n'est limitée que par la mémoire de la machine, et qu'on peut ajouter très rapidement un élément au milieu de la liste.

Si vous savez exactement la taille de votre problème, et que vous ne devez pas déplacer les structures, utilisez les tableaux, c'est le plus commode. S'il y a beaucoup de mouvement de données et que la taille maximale du problème a une limite connue et pas trop grande, utilisez des tableaux de pointeurs. Dans les autres cas, utilisez des listes.

Une dernière solution est bien sûr d'utiliser des tableaux de structures dont on va modifier la taille au fur et à mesure des besoins dans le programme mais cette façon de procéder est peu utilisée car elle est plus lourde que l'utilisation des listes chaînées.

11 Les constructions avancées en C

11.1 Les fonctions récursives

Une caractéristique très pratique du langage C est que les fonctions peuvent s'appeler elle-mêmes. Cela s'appelle la *récursion*. Ceci permet d'écrire des programmes très élégants (même s'ils ne sont pas toujours les plus rapides). Considérons par exemple la factorielle qui peut être définie de la façon suivante: 1! = 1, x! = x.(x-1)! (Factorielle de 1 = 1, et factorielle de x égale x fois factorielle de x moins 1). On peut écrire le programme de la façon suivante:

```
#include <stdio.h>

int fact (int x)
{
  printf ("Computing fact %d\n", x) ;
  if (x == 1)
    return 1 ;
  else
    return x * fact (x-1) ;
}

main ()
{
  printf ("5! = %d\n", fact (5)) ;
}
```

Programme 87

Exercice 96. *La suite de Fibonacci commence par les nombres 1, 1. Chaque nombre suivant est la somme des deux précédents. La suite est donc 1, 1, 2, 3, 5, 8, 13, 21,... Ecrivez un programme récursif qui calcule le* $n^{ième}$ *nombre de Fibonacci.*

Exercice 97. *Dans le programme de l'exercice 96, vous effectuez très certainement une double récursion (Fib(n) = Fib(n-1)+Fib(n-2), ce qui fait que vous calculez plusieurs fois chaque terme de la suite. Pouvez-vous modifier votre programme pour n'effectuer qu'une simple récursion?*

11.2 Le type union

Il est quelquefois nécessaire de mémoriser dans une variable des valeurs dont le type varie selon les circonstances. Ceci peut être fait grâce au type union. Un exemple tiré de *The C Programming Language:*

```
typedef union uT
{
  int ival ;
  float fval;
  char *sval;
} u;
```

Programme 88

La variable `u` déclarée avec le type `union uT` peut contenir un entier, un réel, ou un pointeur vers une chaîne de caractères, selon la nécessité. Le compilateur se charge de la gestion de l'espace mémoire, mais le programmeur doit lui même se souvenir ce qu'il a mis dans `u`. `u` contient la dernière valeur attribuée, et si elle est lue en tant qu'un autre type, cela peut poser des problèmes. De même que pour les structures, l'accès à une variable dans l'`union u` se fait par `u.ival` ou `u.fval`. Encore une fois, si on écrit `u.ival = 2;` et puis `n=strlen(u.sval);`, on peut s'attendre à tout en particulier un plantage!

Comme exemple, nous allons utiliser plus subtilement l'union u:

```
union uT u;

u.sval = "chaine de char";
printf ("l'addresse est %d", u.ival);
```

Programme 89

Ici, on mémorise d'abord l'adresse de `"chaine de char"` dans u, vu comme pointeur vers une chaîne de caractères. Ensuite, on traite u vu comme entier, ce qui imprime la vraie valeur de l'adresse mémoire de `"chaine de char"`. (Remarque: écrire `printf ("%d", u.sval);` a le même effet)

Pour l'exemple ci-dessus, on voulait voir la même valeur sous différentes formes. Dans certains cas on veut se rappeler si l'on avait mémorisé un entier ou un réel par exemple pour pouvoir les relire sous la forme dans laquelle ils avaient été mémorisés. Pour cela il faut ajouter une variable de plus qui indique le type effectivement contenu dans l'union. On affecte alors à cette variable des valeurs différentes lorsque l'on dépose différents types de valeurs dans l'union. Par exemple on peut décider qu'elle est mise à 1 lorsque l'on a mémorisé des entiers et à 2 lorsque l'on a mémorisé des réels, 3 si u contient un `char *`.

En pratique, on ne se sert des unions que pour des problèmes extrêmement pointus donc rares.

12 Compilation séparée et programmation modulaire

12.1 Compilation séparée

Dès que les programmes dépassent une certaine taille, il devient difficile de les manipuler dans un seul fichier. On va au contraire chercher à regrouper les fonctions en groupes de fonctions ayant des rapports entre elles et on va utiliser un fichier source différent pour chacun de ces groupes, ce qui permet de mieux organiser l'ensemble du code source.

Fichiers d'en-tête. Observons le programme suivant:

```
/* premier fichier : princ.c */
long carre (long x) ;
main ()
{
  long y;
  y := carre (15);
}
```

```
/* second fichier : calculs.c */
long carre (long x)
{
  return x * x;
}
```

Programme 91

Pour pouvoir appeler une fonction définie dans un autre fichier il faut qu'elle ait été déclarée auparavant. Par exemple, la fonction principale `main` précédente utilise la fonction `carre` qui est définie dans le fichier `calculs.c`. Pour pouvoir faire cet appel, on a rajouté la déclaration de la fonction `carre` au début du fichier `princ.c`, avant d'effectuer l'appel à la fonction. La présence de cette déclaration permet au compilateur de vérifier que la fonction est correctement appelée, c'est-à-dire avec le bon nombre et le bon type de paramètres. Il faut bien entendu que la déclaration de la fonction soit la même dans le fichier de définition et dans le programme principal.

Le programme peut alors être compilé en un programme exécutable, `essai`, au moyen de la commande:

```
gcc -g -o essai princ.c calculs.c
```

12.2 Bibliothèques de fonctions (librairies)

Que se passe-t-il si l'on a plusieurs programmes qui utilisent, par exemple, les routines d'initialisation et d'affichage de matrice? On peut bien entendu faire du copier-coller à chaque fois. Cela pose cependant un problème. Que faire lorsque l'on détecte une erreur dans une des routines de calcul matriciel? Effectuer les changements dans tous les programmes où les fonctions ont été copiées serait la source de beaucoup d'autres erreurs.

Pour remédier à cela, on préfère extraire les routines suffisamment générales pour être susceptibles d'être utilisées dans plusieurs programmes, et les regrouper dans des fichiers séparés. C'était par exemple, le cas des routines matricielles présentées à la section 7.3. Ces routines ont été écrites originellement dans deux fichiers `matrix.c` et `vector.c` dont un extrait est présenté dans le programme 92.

```
/* matrix.c (extrait) */
...

MatrixPT CreateMatrix (unsigned int lines, unsigned int cols)
{
    MatrixPT m;
    unsigned int i;

    if ((m = (MatrixPT)malloc(sizeof(MatrixT)))==NULL) {
        printf("\nERROR: Can't allocate requested matrix (malloc error).\n");
        return NULL;
    }
    m->lines = lines;
    m->cols  = cols;
    if ((m->val = (double *)malloc(lines*cols*sizeof(double))) == NULL) {
        printf("\nERROR: Can't allocate requested matrix (malloc error).\n");
        free(m);
        return NULL;
    }

    for (i=0; i<lines*cols; i++)
        m->val[i] = 0.0;

    return m;
}
...
```

Programme 92

Pour éviter les erreurs et ne pas avoir à recopier les déclarations des fonctions définies dans `matrix.c` et `vector.c` dans tous les fichiers qui en font usage, on regroupe ces déclarations dans un fichier, dit *fichier d'en-tête* (*header file*), d'extension `.h`, `matrix.h` (programme 40). Ce fichier est alors inclus au moyen de la directive:

```
#include "matrix.h"
```

dans les fichiers qui appellent des fonctions qui y sont déclarées.

12.3 Compilation, édition de liens et Makefile

Si l'on observe la commande de compilation du programme 91:

```
gcc -g -o essai princ.c calculs.c
```

on constate que l'on recompile à chaque fois les deux fichiers princ.c et calculs.c. Sur le même modèle, la ligne de commande pour compiler un fichier `testmat.c` utilisant les routines matricielles définies dans `vector.c` et `matrix.c` serait:

```
gcc -g -o testmat testmat.c vector.c matrix.c
```

Supposons que les fichiers `matrix.c` et `vector.c` soient grands, et ne changent plus beaucoup (ils contiennent beaucoup de routines de calcul matriciel, qui fonctionnent correctement). Supposons que le fichier `testmat.c` soit relativement court, et qu'il change souvent (c'est le programme que l'on essaie d'écrire et il

contient des fautes). Chaque fois qu'on lance la commande de compilation précédente, on recompile les trois fichiers. C'est du temps perdu, vu que les fichiers `matrix.c` et `vector.c` ne changent pas d'une fois à l'autre.

Compilation et édition de liens. Pour comprendre ce qui suit, il faut savoir que ce que nous avons appelé compilation jusqu'à présent comporte en fait deux étapes: la *compilation* proprement dite, et l'*édition de liens*. La compilation proprement dite transforme un programme C en langage machine, sans se soucier des appels de procédure. L'éditeur de liens se charge seulement des appels de procédure, c'est-à-dire, il vérifie que toutes les procédures qui sont appelées ont bien été déclarées, et lie les appels de procédure à leur déclaration.

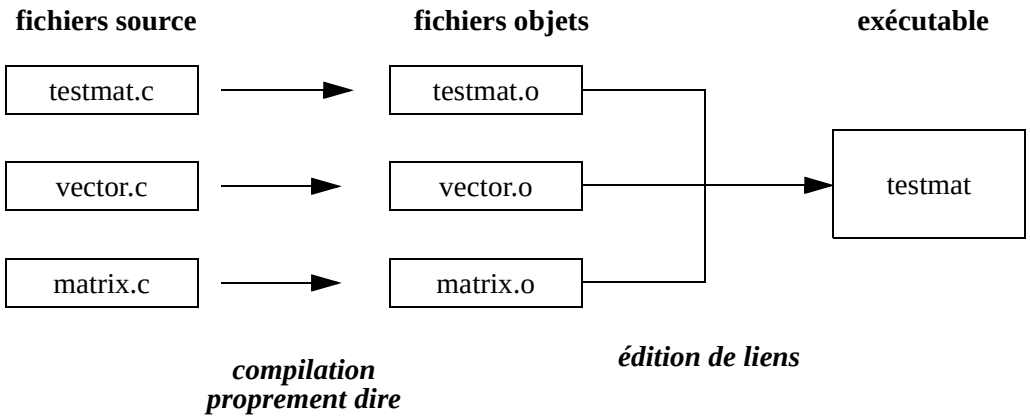


FIGURE 26: Les étapes de compilation

La compilation proprement dite transforme un *fichier source* en un *fichier objet*. L'édition de liens transforme une liste de fichiers objets en un fichier exécutable (figure 26). Par défaut, le compilateur effectue à la fois la compilation proprement dite et l'édition de liens. L'option de compilation `-c` demande au compilateur de ne pas effectuer l'édition de lien.

```
                option de compilation -c qui demande au compilateur de
                ne pas effectuer l'édition de lien
cosun12% cc -c -g vector.c
cosun12%          commande de compilation proprement dite
cosun12%          du fichier vector.c en un fichier object
cosun12%
cosun12% cc -c -g matrix.c
cosun12%          commande de compilation proprement dite
cosun12%          du fichier matrix.c en un fichier object
cosun12%
cosun12% cc -c -g testmat.c
cosun12%          commande de compilation proprement dite
cosun12%          du fichier testmat.c en un fichier object
cosun12%
cosun12% cc -g -o testmat testmat.o matrix.o vector.o
cosun12%          commande d'édition de lien entre les fichiers
                  testmat.o, matrix.o et vector.o en un fichier exécutable
```

FIGURE 27: Les commandes de compilation

La figure 27 décrit les trois commandes qui permettent de faire la compilation des trois fichiers source `testmat.c`, `matrix.c` et `vector.c` en fichiers objets, et la commande d'édition de lien. La première commande effectue la compilation proprement dite du fichier `vector.c` en un fichier objet `vector.o`. Le compilateur n'effectue pas l'édition de liens à cause de l'option de compilation `-c`. La deuxième commande transforme le fichier source `matrix.c` en un fichier objet `matrix.o`. La troisième commande fait de même

avec le fichier `testmat.c`. Enfin, la quatrième effectue l'édition de lien sur les fichiers objet `matrix.o`, `vector.o` et `testmat.o` et produit un fichier exécutable appelé `testmat`.

Pour raccourcir cela, on peut utiliser l'une des commandes suivantes:

```
cosun12% cc -c -g matrix.c
cosun12% cc -c -g vector.c
cosun12%
cosun12% cc -g -o testmat testmat.c matrix.o vector.o
```

FIGURE 28: Les commandes de compilation (2)

Les deux premières commandes transforment les fichiers source `matrix.c` et `vector.c` en fichiers objets `matrix.o` et `vector.o`. La deuxième commande effectue la compilation proprement dite du fichier `testmat.c`, le transforme en un fichier objet et effectue directement l'édition de liens entre ce fichier objet et les deux autres `matrix.o` et `vector.o` pour produire un fichier exécutable appelé `testmat`. En cas de modification du fichier `testmat.c`, on peut recompiler en utilisant la troisième commande de la figure 28.

Bibliothèque de fonctions (librairie). On pourrait en rester là et compiler d'autres programmes se servant des routines de calcul matriciel comme `testmat` en ajoutant `matrix.o` et `vector.o` à la fin de la ligne de compilation. Mais imaginons que les routines matricielles aient été définies non pas en deux fichiers mais en vingt, cela donnerait des lignes de compilation beaucoup trop longues et fastidieuses à taper, sans parler des problèmes d'organisation des fichiers. En fait, les fichiers objets peuvent être regroupés en un seul fichier dans ce que l'on appelle une *bibliothèque* (ou *librairie* par mauvaise traduction de l'anglais *library*), c'est ce que l'on a fait à partir des fichiers objet `matrix.c` et `vector.c` pour créer la bibliothèque `libmatrix.a` présentée à la section 7.3. La commande Unix rassemblant des fichiers objet dans une librairie est `ar`. Voici un exemple d'appel de cette commande:

```
cosun12% ar cvq libmatrix.a matrix.o vector.o
```

Dès lors, pour compiler `testmat.c`, plutôt que de mentionner explicitement `matrix.o` et `vector.o` à la fin de la ligne de commande de compilation, on peut utiliser l'option `-l` du compilateur servant à effectuer l'édition de lien avec une librairie:

```
cosun12% cc -g -o testmat testmat.c -lmatrix
```

Notez que l'on doit omettre le préfixe `lib` ainsi que le suffixe `.a` du nom de la librairie en argument de l'option `-l`, c'est pourquoi pour lier `testmat` avec `libmatrix.a` on utilise `-lmatrix`.

Makefile. Lorsqu'un programme contient beaucoup de fichiers, il est parfois assez difficile de se rappeler quel fichier il faut recompiler. Il existe un utilitaire très pratique, appelé `make`, qui utilise un fichier de dépendances appelé `Makefile` qui décrit quels fichiers source il faut pour obtenir un exécutable donné. Grâce à cette description `make` est en mesure de recompiler seulement les fichiers qui ont été modifiés depuis la dernière compilation.

Un fichier `Makefile` se présente comme suit (figure 29):

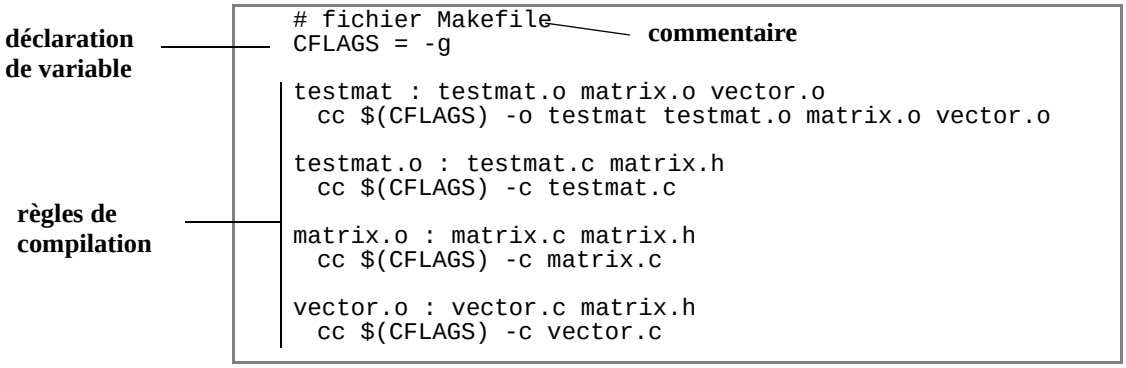


FIGURE 29: Le fichier Makefile

Dans un Makefile, les lignes en commentaires commencent par le caractère # (dièse, *pound*). Les variables, comportent par convention exclusivement des caractères majuscules et _ et sont définies en utilisant la syntaxe `NOM_DE_VARIABLE = valeur`. La deuxième ligne du fichier `Makefile` précédent définit par exemple une variable `CFLAGS` (option de compilation), qui prend la valeur `-g`. Dans ce cas, l'intérêt de définir une variable est de pouvoir changer les options de compilation C pour tous les fichiers d'un seul coup. Tant que l'on n'est pas sûr du fonctionnement de son programme, on conserve l'option `-g` pour pouvoir utiliser le debugger. Une fois que le programme fonctionne, on remplace l'option `-g` par l'option `-O` (optimisation), pour améliorer la performance du programme.

Après les déclarations de variables viennent les règles de compilation. Les règles de compilation comportent 3 parties:

- le nom du fichier que l'on souhaite créer (appelé fichier *cible*, *target*) suivi de '`:`'
- la liste des fichiers dont le fichier cible dépend (les *dépendances, dependencies*). Toute modification d'un de ces fichiers entraîne la régénération du fichier cible
- A la ligne suivante: un caractère de tabulation TAB suivi de la commande qui permet de régénérer le fichier cible à partir de ses dépendances. **ATTENTION**: une ligne de commande dans un fichier Makefile doit impérativement commencer par une tabulation (pas des espaces).

Par exemple, la première règle dit que le fichier exécutable `testmat` dépend des fichiers objets `testmat.o`, `matrix.o` et `vector.o`. Pour reconstituer le fichier exécutable à partir des fichiers `testmat.o`, `matrix.o` et `vector.o`, on utilise la commande `cc $(CFLAGS) -o testmat testmat.o matrix.o vector.o`. Dans cette commande, la notation `$(CFLAGS)` veut simplement dire: le contenu de la variable `CFLAGS`, en l'occurrence "`-g`". La deuxième règle dit que le fichier `testmat.o` dépend des fichiers `testmat.c` et `matrix.h`. Pour reconstituer le fichier `testmat.o`, on utilise la commande `cc $(CFLAGS) -c testmat.c`. On remarque que les commandes utilisées sont exactement les commandes de la figure 27.

Pour utiliser le fichier de dépendances, on utilise la commande:

```
cosun12% make
cosun12%
```

L'avantage de cette manipulation est qu'en utilisant la commande `make`, on ne recompilera que les fichiers qui ont été modifié depuis la dernière compilation. Pour obtenir ce résultat, la commande `make` vérifie la date de création des différents fichiers auxquels il est fait référence dans le fichier `Makefile`. La commande procède comme suit. Lors de son invocation, si aucune cible n'est précisée, `make` tente de générer la première cible se trouvant dans le fichier `Makefile`, en l'occurrence `testmat`. `make` considère alors les dépendances de `tesmat` (`testmat.o`, `matrix.o` et `vector.o`) l'une après l'autre pour vérifier s'il existe des règles pour chacune d'elles. C'est par exemple le cas pour `tesmat.o`. `make` tente donc de générer `testmat.o` d'après sa règle, pour cela `make` considère d'abord les dépendances de `testmat.o` (`testmat.c` et `matrix.h`) l'une

après l'autre pour vérifier s'il existe des règles pour les générer. Or il n'y en a pas, `testmat.c` et `matrix.h` sont donc considérés comme des fichiers terminaux qui ne peuvent être générés. Dans ce cas `make` compare leur date de modification à celle de la dernière génération de `testmat.o`. Si `testmat.o` est plus récent que la dernière modification de `testmat.c` et `matrix.h`, alors il est à jour et ce n'est pas la peine de le régénérer. Dans le cas contraire la règle de compilation `cc $(CFLAGS) -c testmat.c` est appliquée. `make` procède de même pour les autres dépendances matrix.o et vector.o. Si l'une au moins des dépendances est plus récente que sa cible alors la cible est régénérée au moyen de la règle de compilation.

Notons qu'il est possible de raccourcir le fichier Makefile en utilisant ce que l'on appelle des règles de compilation *génériques*, d'effectuer de la compilation de fichiers sous conditions, etc... Les possibilités offertes par `make` sont énormes et dépassent le cadre de ce cours.

13 Routines graphiques avancées

Les routines graphiques introduites dans la section 7.4 ne permettent pas d'interagir avec la fenêtre graphique, comme on souhaiterait le faire avec un jeu. Les deux sections suivantes discutent deux modes d'interaction possible avec un programme graphique: (1) par la fenêtre terminal; (2) par la fenêtre graphique elle même.

13.1 Interaction dans la fenêtre terminal

Le fichier `Graphics.h` contient quatre routines supplémentaires qui permettent d'interagir avec la fenêtre graphique, par l'intermédiaire de la fenêtre terminal dans laquelle vous avez lancé le programme.

```
void StartSingleCharacterMode (void) ;
void FinishSingleCharacterMode (void) ;
void GetSingleCharacter (char c) ;
void CheckForSingleCharacter (char c) ;
```

Programme 93

Le mode d'interaction par la fenêtre terminal permet au programme de réagir à chaque touche de clavier enfoncée. C'est différent de l'interaction habituelle avec l'instruction C `scanf` ou `getchar`. Dans ces cas, le programme ne réagit que lorsqu'on enfonce la touche `Return`.

Pour pouvoir utiliser ces fonctions, il faut au début du programme appeler la procédure `StartSingleCharacterMode`, qui permet au programme de réagir immédiatement, dès qu'une touche est enfoncée. Le programme est fait d'une boucle qui affiche quelque chose dans la fenêtre graphique, attend un petit peu, demande à l'utilisateur ce qu'il faut faire, et recommence la boucle. Pour interroger l'utilisateur, le programmeur dispose de deux routines: `GetSingleCharacter` et `CheckForSingleCharacter`. `GetSingleCharacter` bloque le programme jusqu'à ce que l'utilisateur enfonce une touche. `CheckForSingleCharacter` retourne '`?`' si l'utilisateur n'enfonce pas de touche. Si l'utilisateur a enfoncé une touche, la fonction retourne la lettre correspondant à la touche enfoncée. A la fin du programme la routine `FinishSingleCharacterMode` doit être appelée. Entre les appels `StartSingleCharacterMode` et `FinishSingleCharacterMode`, il est interdit d'employer les routines `scanf` ou `getchar`.

Les programmes `~gennart/exercice-corriges/g3.c` et `~gennart/exercice-corriges/g4.c` illustrent l'utilisation des 4 fonctions du programme 93. Les interactions sont limitées aux lettres de l'alphabet (majuscules et minuscules) et aux chiffres. Vous ne pouvez pas, par exemple, utiliser les flèches de contrôle entre le clavier et le pavé numérique, ni la souris. Cependant ces programmes illustrent bien le concept de boucle d'événements. Le corps des deux programmes est une boucle infinie qui agit en fonction des touches enfoncées par l'utilisateur.

13.2 Interaction dans la fenêtre graphique

Les routines de la section précédente ne permettent pas d'utiliser la souris pour les interactions avec la fenêtre graphique. Pour arriver à ce résultat, il faut changer complètement de modèle de programmation. Dans les

programmes `g3.c` et `g4.c` de la section précédente, il y avait une boucle infinie qui traitait les interventions de l'utilisateur au clavier. Maintenant, la boucle d'événements est cachée dans la librairie, et le programmeur la lance en invoquant la procédure `GraphicsLoop`. Cette procédure comporte deux paramètres: le nom de la routine qui est exécutée à chaque itération de la boucle, et le délai entre chaque itération de la boucle. La routine exécutée à chaque itération de la boucle rafraîchit en général l'écran en fonction de l'état du programme.

Avant de lancer la procédure `GraphicsLoop`, il faut indiquer à quels événements la boucle sera sensible. Ces événements peuvent être un mouvement de la souris (`SetMouseMotionRoutine`), l'enfoncement d'une touche du clavier (`SetKeyDownRoutine`), le relâchement d'une touche (`SetKeyUpRoutine`), l'enfoncement d'une touche du clavier en même temps que la touche `Control` (`SetControlKeyDownRoutine`), le relâchement d'une touche pendant que la touche `Control` est encore enfoncée (`SetControlKeyUpRoutine`), l'enfoncement d'un bouton de la souris (`SetButtonDownRoutine`) et le relâchement d'un bouton de la souris (`SetButtonDownRoutine`).

```
enum ButtonT { LeftButton, CenterButton, RightButton };

void SetMouseMotionRoutine (void MouseMotion (int x, int y)) ;
void SetKeyDownRoutine (char c, void KeyRoutine ()) ;
void SetKeyUpRoutine (char c, void KeyRoutine ()) ;
void SetControlKeyDownRoutine (char c, void KeyRoutine ()) ;
void SetControlKeyUpRoutine (char c, void KeyRoutine ()) ;
void SetButtonDownRoutine (int ButtonT, void ButtonRoutine ()) ;
void SetButtonUpRoutine (int ButtonT, void ButtonRoutine ()) ;

void PrintMouseEvent (int x, int y) ;
void PrintKeyEvent () ;
void PrintButtonEvent () ;

void DoNothing () ;
void GraphicsLoop (void repeatRoutine (), int delay) ;
void ToggleDebugMode () ;
```

Programme 94

La routine `ToggleDebugMode` active ou désactive les messages d'informations de la librairie d'interaction graphique. Les programmes `~gennart/exercice-corriges/g5.c` et `~/gennart/exercice-corriges/g6.c` illustrent le fonctionnement des procédures du programme 94.

14 Exercices avancés

Tous les exercices que nous avons vus jusqu'à présent ont servi à vous faire comprendre les constructions du langage C. Dans cette section, nous essayons de résoudre des problèmes pratiques, en utilisant les constructions du langage.

Ces exercices représentent une étape assez importante. Jusqu'à présent, le but était d'écrire des programmes courts. A partir ce cette section, avant d'écrire votre programme, il faut transformer un énoncé en français (expliquant un problème de balistique, la résolution des contraintes dans une structure, ou la rotation en 3-D) en une séquence d'opérations à décrire en C. Les deux étapes: *conception* du programme, c'est-à-dire, la transformation d'un énoncé en une séquence d'opérations et la *réalisation* du programme, c'est-à-dire la traduction de ces opérations en C ont autant d'importance l'une que l'autre.

Cependant, dans cette section, nous insisterons surtout la partie conception, qui est nouvelle, et laisserons la réalisation des programmes comme exercice.

14.1 Méthode de Runge-Kutta

Explication théorique. Un certain nombre de problèmes de mécanique du mouvement consistent à résoudre de proche en proche une équation différentielle. Les données du problème sont la valeur d'une fonction au temps t_0, $x(t_0)$, et la valeur de la dérivée de la fonction en fonction du temps, $\dot{x}(t) = f(t, x(t))$. x peut être un scalaire ou un vecteur. La valeur de la dérivée est une fonction de la valeur de x au temps t. Une première

façon de calculer la fonction *x* au temps $t_i = t_{i-1} + \Delta t$, en d'utiliser la formule de l'équation (1), dite formule d'Euler.

$$x(t + \Delta t) \approx x(t) + \Delta t \cdot \dot{x}(t) = x^{EU}(t + \Delta t) \quad \text{(EQ 1)}$$

Dans cette expression, $x(t + \Delta t)$ représente la valeur exacte de la fonction *x* au temps *t*+Δ*t*, et $x^{EU}(t + \Delta t)$ la valeur approchée de la fonction *x* au temps *t*+Δ*t*, en utilisant la formule de l'équation (1). Graphiquement, l'équation (1) se représente de la façon suivante:

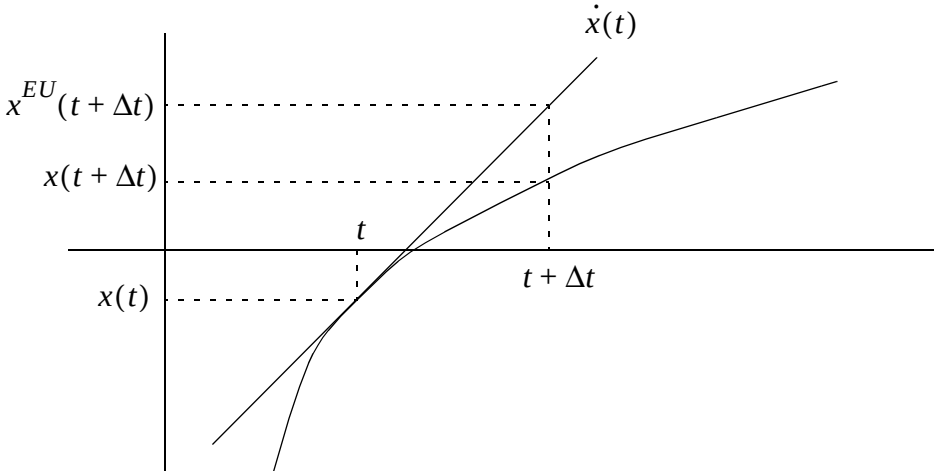


FIGURE 30: Illustration de la formule d'Euler

On constate donc que l'on obtient une valeur approchée de la valeur *x(t)*. Comme on fait les calculs de proche en proche, les erreurs s'accumulent. On constate aussi qu'il vaut mieux ne pas choisir une valeur trop grande du pas Δ*t*, sinon l'erreur devient très grande très rapidement.

La méthode de Runge-Kutta d'ordre 2 permet de calculer de proche en proche la valeur d'une fonction, de façon beaucoup plus précise que l'approximation de la formule (1). L'idée est de prendre la dérivée au point t+Δt/2 et de trouver la valeur approchée $x(t + \Delta t)$ selon la formule:

$$x^{\mathrm{RK2}}(t + \Delta t) = x(t) + \Delta t \cdot \dot{x}(t + \Delta t/2) \quad \text{(EQ 2)}$$

Le problème est qu'on ne connaît pas la valeur de $\dot{x}(t + \Delta t/2)$. Pour évaluer la formule (2), trois étapes:

- on calcule $\dot{x}(t) = f(t, x(t))$
- on évalue d'abord un première approximation de $x(t + \Delta t/2)$, en utilisant la formule de l'équation (1). ceci nous donne l'expression $x^{EU}(t + \Delta t/2)$
- on calcule ensuite $\dot{x}(t + \Delta t) = f(t + \Delta t/2, x^{EU}(t + \Delta t/2))$, une estimation de la dérivée en $t + \Delta t$.
- On a alors toutes les valeurs nécessaire pour évaluer la formule de l'équation (2).

L'interprétation graphique de la formule (2) est:

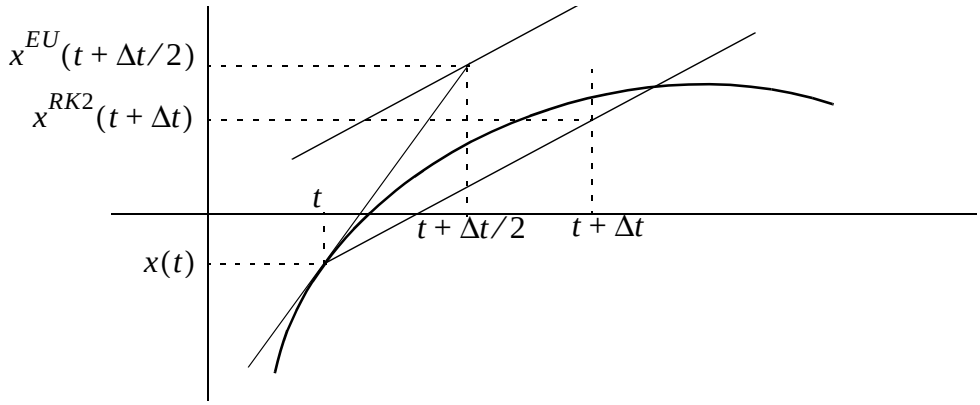


FIGURE 31: Méthode de Runge-Kutta d'ordre 2

On peut faire mieux encore, avec une méthode Runge-Kutta d'ordre 4. La série de formules permettant de calculer la nouvelle valeur de $x^{\mathrm{RK4}}(t + \Delta t)$ est:

$$\begin{aligned} k_1 &= \Delta t \cdot f(t, x(t)) \\ k_2 &= \Delta t \cdot f(t + \Delta t/2, x(t) + k_1/2) \\ k_3 &= \Delta t \cdot f(t + \Delta t/2, x(t) + k_2/2) \\ k_4 &= \Delta t \cdot f(t + \Delta t, x(t) + k_3) \\ x^{\mathrm{RK4}}(t + \Delta t) &= x(t) + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} \end{aligned} \qquad \text{(EQ 3)}$$

Quantité x(t) scalaire: tracé d'une courbe du 3ème degré. On considère $f(x(t)) = 3t^2-1$, et $t_0 = -2$, $x(t_0) = -6$. On demande de tracer en fonction de t:

- la courbe t^3-t entre -2 et 2,
- la courbe qui résulte de l'évaluation de la formule (1), avec des pas $\Delta t = 0.25$,
- la courbe qui résulte de l'évaluation de la formule (2), avec le même pas
- la courbe qui résulte de l'évaluation des formules (3).

Le fichier `degre3.c` sur le serveur CO contient les trois premières parties.

Runge-Kutta avec une quantité x(t) vectorielle: balistique. La trajectoire d'un projectile est affectée par la gravitation et le frottement de l'air. La force due à la gravitation s'écrit $\vec{F_g} = \vec{g} \cdot m$. La force due au frottement est proportionnelle à la vitesse et s'écrit $\vec{F_f} = k \cdot \vec{V}$. L'équation du mouvement est donc:

$$\begin{aligned} \vec{F} &= m\vec{a} \\ \vec{F_g} + \vec{F_f} &= m\vec{a} \\ \vec{g} \cdot m - k \cdot \vec{V} &= m\vec{a} \\ \begin{pmatrix} 0 \\ -g \end{pmatrix} \cdot m - \begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix} \cdot k &= \begin{pmatrix} \ddot{x} \\ \ddot{y} \end{pmatrix} \cdot m \end{aligned} \qquad \text{(EQ 4)}$$

Pour avoir une forme canonique, il faut introduire deux variables supplémentaires v_x et v_y. Le système devient alors:

$$\begin{aligned} \dot{x} &= v_x \\ \dot{y} &= v_y \\ \ddot{x} = \dot{v}_x &= 0 - \left(v_x \cdot \frac{k}{m} \right) \\ \ddot{y} = \dot{v}_y &= -g - \left(v_y \cdot \frac{k}{m} \right) \end{aligned} \qquad \text{(EQ 5)}$$

Ce système d'équation est de la forme $\dot{\vec{X}}(t) = f\left(\vec{X}(t)\right)$, avec $X = (x, y, v_x, v_y)$. On peut donc appliquer la méthode de Runge-Kutta sur chacun des éléments du vecteur, en utilisant par exemple comme valeurs initiales $(x, y, v_x, v_y) = (0,0,2,0.5)$, $\Delta t = 0.1$, $k = 0.2$, $g=0.1$, et $m=1$. Pour afficher le résultat, on affiche les points $(x(t),y(t))$ pour toutes les valeurs de t calculées.

Runge-Kutta avec une quantité x(t) vectorielle: mouvement de la lune autour de la terre. La force d'attraction entre deux corps est inversement proportionnelle à la distance qui les séparent.

Si on fait l'hypothèse que la Terre est fixe au centre de l'écran et que (x,y) représente la position de la lune, on a donc:

$$\vec{F_g} = -\frac{k \cdot m}{x^2 + y^2} \cdot \frac{1}{\sqrt{x^2 + y^2}} \cdot \begin{pmatrix} x \\ y \end{pmatrix} \qquad \text{(EQ 6)}$$

On a donc un système de la forme:

$$\begin{aligned} \vec{F_g} &= m\vec{a} \\ \begin{pmatrix} x \\ y \end{pmatrix} \cdot \frac{1}{\sqrt{x^2 + y^2}} \cdot -\frac{k}{x^2 + y^2} \cdot m &= \begin{pmatrix} \ddot{x} \\ \ddot{y} \end{pmatrix} \cdot m \end{aligned} \qquad \text{(EQ 7)}$$

En ajoutant les variables v_x et v_y, comme à l'exercice précédent, on obtient un système canonique:

$$\begin{aligned} \dot{x} &= v_x \\ \dot{y} &= v_y \\ \ddot{x} = \dot{v}_x &= -x \cdot \frac{k}{(x^2 + y^2)^{1.5}} \\ \ddot{y} = \dot{v}_y &= -y \cdot \frac{k}{(x^2 + y^2)^{1.5}} \end{aligned} \qquad \text{(EQ 8)}$$

On peut résoudre ce système en utilisant la méthode de Runge-Kutta. Pour afficher le résultat, on affiche les points $(x(t), y(t))$ pour toutes les valeurs de t calculées. Comme valeurs initiales, on peut prendre $x=1.0$, $y=0.0$, $v_x = 0.0$, $v_y = 1.25$. Faites attention à ne pas prendre un Δt trop grand. Une bonne valeur de Δt est par exemple 0.01. Si vous utilisez les routines graphiques pour afficher la position (x,y), utilisez comme taille de graphe $(4.0,-8.0,-4.0,2.0)$.

14.2 Rotation 3-D

Routine d'affichage. On considère que l'on a d'une part un graphe, dont les coordonnées sont des nombres réels, et d'autre part un écran dont les coordonnées sont des nombres entiers. Les axes x et y du graphe vont de gauche à droite et de bas en haut respectivement. Les axes x_e et y_e de l'écran vont de gauche à droite et de HAUT en BAS. La coordonnée *(0,0)* de l'écran est au coin supérieur gauche.

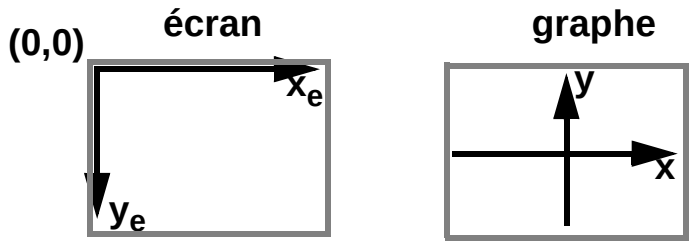


FIGURE 32: Coordonnées de graphe et d'écran

Pour dessiner un graphe, l'utilisateur spécifie d'abord la taille de l'écran et la taille du graphe. Il dessine ensuite une série d'objets (points, vecteurs,...) dont les points sont spécifiés en coordonnées de graphe. Les routines de dessin se chargent automatiquement de transformer coordonnées de graphe en coordonnées d'écran.

Pour éviter des routines d'affichage ayant de trop nombreux paramètres, on suggère de déclarer dans le module des variables globales représentant les tailles d'écran et de graphe. Deux routines permettent d'initialiser les variables de taille d'écran et de graphe. D'autre part, les deux routines d'affichage permettent d'afficher des points et des vecteurs. Les coordonnées des points passés en paramètre aux routines sont des coordonnées de graphe. Le paquetage graphique doit donc contenir les informations suivantes:

```
void SetGraphSize(float top,float left,float bottom,float right);
void SetScreenSize (int top, int left, int bottom, int right) ;
void DrawPoint (float x, float y, float diameter) ;
void DrawVector (float fromX, float fromY, float toX, float toY);
```

Chacune des deux routines d'affichage est constituée de deux parties. La première transforme les coordonnées de graphe en coordonnées d'écran (`GraphToScreen`). La deuxième affiche à l'écran les données nécessaires. La routine `GraphToScreen` prend en considération la taille du graphe et la taille de l'écran.

Routine de projection d'un point 3-D sur un plan. Les coordonnées 3D sont écrites (X, Y, Z). L'axe X va de l'arrière vers l'avant. L'axe Y va de gauche à droite. L'axe Z va de bas en haut. Les coordonnées 2D sont

écrites (x, y). L'axe x va de gauche à droite. L'axe y va de bas en haut. Pour effectuer la projection, on propose la méthode simplifiée suivante:

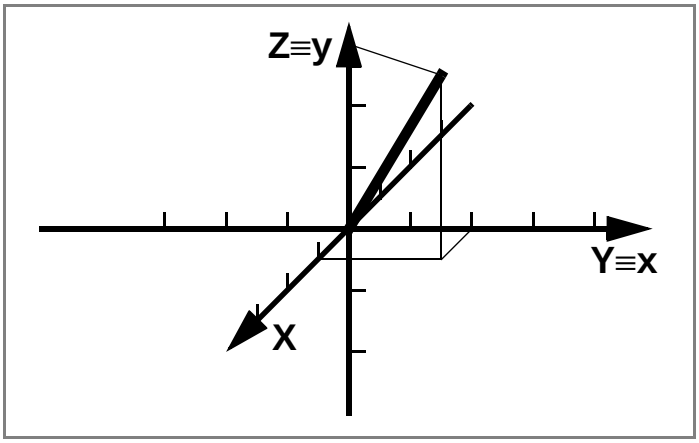


FIGURE 33: Affichage 3-D

Pour afficher un point 3D dont la coordonnée en X est 0, on utilise comme coordonnées (x, y) les coordonnées Y et Z du point 3D. Si la coordonnée X du point 3D est non-nulle, on soustrait de Y et de Z la moitié de la coordonnée X. Sur le graphe ci-dessus, l'extrémité du vecteur (1, 2, 3) est affichée en 2D à la coordonnée (2-0.5 =1.5, 3-0.5 = 2.5). En notation matricielle, l'opération de projection est écrite:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} -0.5 & 1 & 0 \\ -0.5 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \qquad \text{(EQ 9)}$$

L'opération de projection peut donc être écrite simplement sous forme de produit matriciel. Votre routine de projection devrait donc prendre tout au plus quelques lignes. Vérifier en utilisant les routines d'affichage 2D et la routine de projection que vous arrivez à afficher le graphe ci-dessus.

Rotation d'un vecteur autour d'un axe quelconque. L'axe de rotation et le vecteur passent par l'origine. Le programme demande à l'utilisateur, l'axe de rotation et le vecteur à faire tourner. Il affiche les axes de référence et l'axe de rotation (voir figure 34), puis fait faire deux tours au vecteur autour de l'axe de rotation. L'axe et le vecteur sont spécifiés par un point 3D, par exemple (1, 2, 3).

La rotation autour d'un axe quelconque consiste à ramener, par deux rotations autour des axes de référence Z et Y, l'axe de rotation sur l'axe de référence X, faire tourner le vecteur autour de l'axe X, et ramener l'axe de rotation à sa position originale par deux rotations autour des axes de référence Y et Z.

On écrira d'abord 3 routines d'initialisation de matrice (3,3) de rotation autour des 3 axes de référence (faites attention, les routines `sin` et `cos` prennent comme paramètre des radians):

```
void InitXRotationMatrix(MatrixPT m, double theta);
void InitYRotationMatrix(MatrixPT m, double theta);
void InitZRotationMatrix(MatrixPT m, double theta);
```

Ces trois matrices contiennent les valeurs suivantes:

X axis	Y axis	Z axis
$\begin{bmatrix} 1.0 & 0.0 & 0.0 \\ 0.0 & \cos\theta & -\sin\theta \\ 0.0 & \sin\theta & \cos\theta \end{bmatrix}$	$\begin{bmatrix} \cos\theta & 0.0 & \sin\theta \\ 0.0 & 1.0 & 0.0 \\ -\sin\theta & 0.0 & \cos\theta \end{bmatrix}$	$\begin{bmatrix} \cos\theta & -\sin\theta & 0.0 \\ \sin\theta & \cos\theta & 0.0 \\ 0.0 & 0.0 & 1.0 \end{bmatrix}$

Faire tourner un vecteur d'un angle θ autour d'un axe dont les deux angles sont α et β consiste à multiplier ce vecteur par cinq matrices de rotation: M0 (rotation d'un angle -α autour de l'axe Z), M1 (rotation d'un angle -β autour de l'axe Y), M2 (rotation d'un angle θ autour de l'axe X), M3 (rotation d'un angle β autour de l'axe Y), M4 (rotation d'un angle α autour de l'axe Z:

$$rotatedVector = M4 \cdot M3 \cdot M2 \cdot M1 \cdot M0 \cdot vector$$

Il reste à déterminer les angles des rotations à effectuer autour des axes Y et Z. Ceci se fait par une transformation des coordonnées cartésiennes (X,Y,Z) de l'axe de rotation en coordonnées sphériques *(r,* α*,* β*)*. Graphiquement, les angles sont définis comme suit:

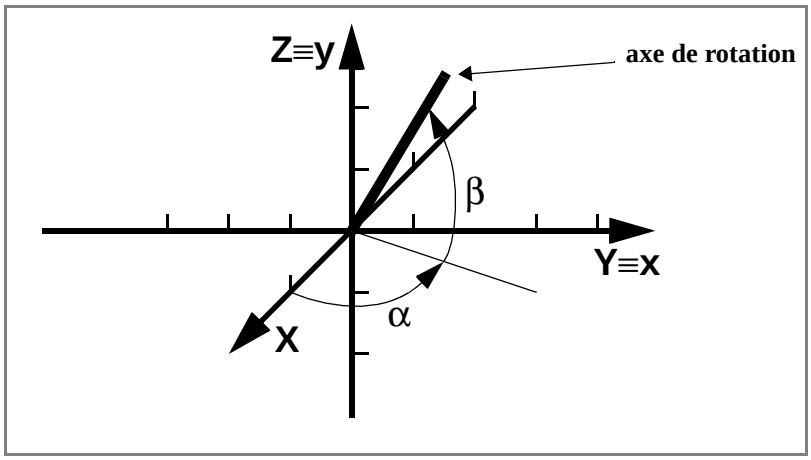


FIGURE 34: Rotation 3D

Les formules pour effectuer cette transformation s'écrivent:

$$r = \sqrt{x^2 + y^2 + z^2}$$

$$\psi = \operatorname{acos}\left(\frac{z}{r}\right) \qquad \psi \in \left[\frac{\Pi}{2};\frac{\Pi}{2}\right]$$

$$\beta = \frac{\pi}{2} - \psi \qquad \beta \in [0;\Pi]$$

$$y \geq 0 \Rightarrow \left(\alpha = \operatorname{acos}\left(\frac{x}{r\sin\psi}\right)\right) \qquad \alpha \in [0;2\Pi]$$

$$y \leq 0 \Rightarrow \left(\alpha = 2\pi - \operatorname{acos}\left(\frac{x}{r\sin\psi}\right)\right) \qquad \alpha \in [0;2\Pi]$$

(EQ 10)

Faites attention, car avec les erreurs d'arrondi, l'expression *z / r* peut être plus grande que 1 (ou plus petite que -1) et prendre l'arccosinus d'une expression plus grande que 1 donne une valeur indéterminée.

Pour ce qui est de l'affichage, la meilleure façon pour donner l'impression de mouvement est de successivement dessiner et effacer le vecteur à différentes positions autour de l'axe de rotation. Comme cela efface des points des axes, il est conseillé de redessiner les axes à chaque étape.

14.3 Calcul des contraintes dans un treillis.

Par définition, le type de treillis que l'on peut résoudre avec l'algorithme présenté contient n noeuds et 2n -3 barres. La figure suivante présente un treillis contenant (1) 2 noeuds et 1 barre; (2) 3 noeuds et 3 barres; (3) 4 noeuds et 5 barres; (4) 5 noeuds et 7 barres. Les treillis plus compliqués se font en ajoutant à chaque étape 1 noeud et 2 barres.

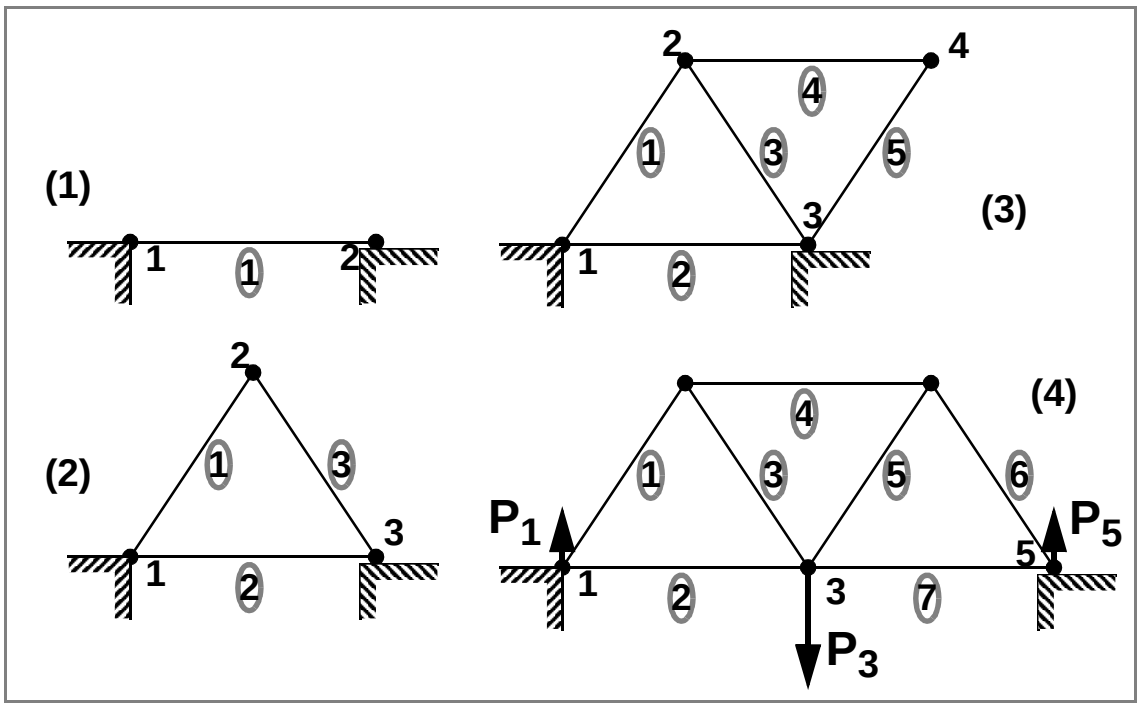


FIGURE 35: Treillis

Le treillis (4) représente un pont, avec deux supports. Un des supports est fixé dans les deux dimensions, l'autre support n'est fixé que verticalement. On applique sur le pont une charge ponctuelle P au noeud P_3 (P_{3x},P_{3y}). A partir de cette charge, il est possible de déterminer les réactions aux supports (R_{1x}, R_{1y}, R_{5y}), en considérant d'une part que la somme des forces en direction des axes X et Y est nulle et d'autre part que la somme des couples appliqués au treillis est nulle, ce qui se traduit par:

$$\begin{aligned} P_{1x} + P_{3x} &= 0 \\ P_{1y} + P_{3y} + R_{5y} &= 0 \\ P_{1y}x_1 + P_{1x}y_1 + P_{3y}x_3 + P_{3x}y_3 + P_{5y}x_5 + P_{5x}y_5 &= 0 \end{aligned} \qquad \text{(EQ 11)}$$

Ce système de trois équations permet de déterminer les réactions aux supports. Le fichier contenant la représentation du treillis contient à la fois les charges et les forces de réactions. Si vous créez un fichier de représentation du treillis, vous devez vous assurer que la somme des forces extérieures appliquées au pont est bien nulle.

Pour déterminer les contraintes dans les barres, on écrit pour chacun des noeuds 2 équations indiquant que la somme des forces selon les 2 axes principaux est nulle. Dans le cas du pont à 5 noeuds, ces équations sont:

noeud 1

$$f1\left(\frac{X_2 - X_1}{|B_{12}|}\right) + f2\left(\frac{X_3 - X_1}{|B_{13}|}\right) + P_{1x} = 0$$

$$f1\left(\frac{Y_2 - Y_1}{|B_{12}|}\right) + f2\left(\frac{Y_3 - Y_1}{|B_{13}|}\right) + P_{1y} = 0$$

noeud 2

$$f1\left(\frac{X_1 - X_2}{|B_{12}|}\right) + f3\left(\frac{X_3 - X_2}{|B_{32}|}\right) + f4\left(\frac{X_4 - X_2}{|B_{42}|}\right) = 0$$

$$f1\left(\frac{Y_1 - Y_2}{|B_{12}|}\right) + f3\left(\frac{Y_3 - Y_2}{|B_{32}|}\right) + f4\left(\frac{Y_4 - Y_2}{|B_{42}|}\right) = 0$$

noeud 3

$$f2\left(\frac{X_1 - X_3}{|B_{13}|}\right) + f3\left(\frac{X_2 - X_3}{|B_{23}|}\right) + f5\left(\frac{X_4 - X_3}{|B_{43}|}\right) + f7\left(\frac{X_5 - X_3}{|B_{53}|}\right) + P_{3x} = 0$$

$$f2\left(\frac{Y_1 - Y_3}{|B_{13}|}\right) + f3\left(\frac{Y_2 - Y_3}{|B_{23}|}\right) + f5\left(\frac{Y_4 - Y_3}{|B_{43}|}\right) + f7\left(\frac{Y_5 - Y_3}{|B_{53}|}\right) + P_{3y} = 0$$

noeud 4

$$f4\left(\frac{X_2 - X_4}{|B_{24}|}\right) + f5\left(\frac{X_3 - X_4}{|B_{34}|}\right) + f6\left(\frac{X_5 - X_4}{|B_{54}|}\right) = 0$$

$$f4\left(\frac{Y_2 - Y_4}{|B_{24}|}\right) + f5\left(\frac{Y_3 - Y_4}{|B_{34}|}\right) + f6\left(\frac{Y_5 - Y_4}{|B_{54}|}\right) = 0$$

noeud 5

$$f6\left(\frac{X_4 - X_5}{|B_{45}|}\right) + f7\left(\frac{X_3 - X_5}{|B_{35}|}\right) + P_{5x} = 0$$

$$f6\left(\frac{Y_4 - Y_5}{|B_{45}|}\right) + f7\left(\frac{Y_3 - Y_5}{|B_{35}|}\right) + P_{5y} = 0$$

(EQ 12)

Chacun des termes:

$$f_{bar}\left(\frac{X_{from} - X_{to}}{|B_{fromto}|}\right) \qquad f_{bar}\left(\frac{Y_{from} - Y_{to}}{|B_{fromto}|}\right)$$

(EQ 13)

représente la projection de la force f_{bar} sur les axes X et Y. Dans ces expressions, le noeud “from” a pour coordonnées (X_{from}, Y_{from}); la longueur d’une barre qui va du noeud “from” au noeud “to” est dénotée $|B_{fromto}|$. Une remarque aussi sur les problèmes de signe. La force dans une barre est considérée positive si elle est en compression. Pour qu’il n’y ait pas d’erreur de signe dans la matrice, il faut veiller à ce que tous les termes X_{to} et Y_{to} sur une même ligne aient le même indice.

Le même système écrit sous forme matricielle est

$$
\begin{array}{l}
\text{noeud 1} \\
\\
\text{noeud 2} \\
\\
\text{noeud 3} \\
\\
\text{noeud 4} \\
\\
\text{noeud 5}
\end{array}
\begin{bmatrix}
\frac{X_2-X_1}{|B_{21}|} & \frac{X_3-X_1}{|B_{21}|} & 0 & 0 & 0 & 0 & 0 \\
\frac{Y_2-Y_1}{|B_{21}|} & \frac{Y_3-Y_1}{|B_{31}|} & 0 & 0 & 0 & 0 & 0 \\
\frac{X_1-X_2}{|B_{12}|} & 0 & \frac{X_3-X_2}{|B_{32}|} & \frac{X_4-X_2}{|B_{42}|} & 0 & 0 & 0 \\
\frac{Y_1-Y_2}{|B_{12}|} & 0 & \frac{Y_3-Y_2}{|B_{32}|} & \frac{Y_4-Y_2}{|B_{42}|} & 0 & 0 & 0 \\
0 & \frac{X_1-X_3}{|B_{13}|} & \frac{X_2-X_3}{|B_{23}|} & 0 & \frac{X_4-X_3}{|B_{43}|} & 0 & \frac{X_5-X_3}{|B_{53}|} \\
0 & \frac{Y_1-Y_3}{|B_{13}|} & \frac{Y_2-Y_3}{|B_{23}|} & 0 & \frac{Y_4-Y_3}{|B_{43}|} & 0 & \frac{Y_5-Y_3}{|B_{53}|} \\
0 & 0 & 0 & \frac{X_2-X_4}{|B_{24}|} & \frac{X_3-X_4}{|B_{34}|} & \frac{X_5-X_4}{|B_{54}|} & 0 \\
0 & 0 & 0 & \frac{Y_2-Y_4}{|B_{24}|} & \frac{Y_3-Y_4}{|B_{34}|} & \frac{Y_5-X_4}{|B_{54}|} & 0 \\
0 & 0 & 0 & 0 & 0 & \frac{X_4-X_5}{|B_{45}|} & \frac{X_3-X_5}{|B_{35}|} \\
0 & 0 & 0 & 0 & 0 & \frac{Y_4-Y_5}{|B_{45}|} & \frac{Y_3-Y_5}{|B_{35}|}
\end{bmatrix}
\cdot
\begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \\ f_7 \end{bmatrix}
=
\begin{bmatrix}
-P_{1x} \\ -P_{1y} \\ -P_{2x} \\ -P_{2y} \\ -P_{3x} \\ -P_{3y} \\ -P_{4x} \\ -P_{4y} \\ -P_{5x} \\ -P_{5y}
\end{bmatrix}
\qquad \text{(EQ 14)}
$$

Les esprits attentifs auront remarqué que le système comporte 10 équations pour 7 inconnues. De par la définition du problème cependant, les trois dernières équations sont des combinaisons linéaires des 7 premières. On peut donc les ignorer, et résoudre un système de 7 équations à 7 inconnues.

D'autres considérations permettent de construire la matrice assez facilement:

- chaque colonne contient 4 éléments non nuls, qui correspondent tous à une barre
- chaque ligne contient la projection sur un axe (vertical ou horizontal) de la force interne des barres qui aboutissent à un noeud donné.

14.4 Le labyrinthe

On se propose de modéliser un appartement comportant plusieurs chambres en utilisant la structure `ChambreT`:

```
typedef struct ChambreT
{
  char nom[32] ;
  int index ;
  int c1, c2, c3 ;
} ChambreT;
```

Programme 95

Le champ `nom` contient le nom de la chambre, le champ `index` le numéro de la chambre. Chacune des chambres comporte 3 portes. Les champs `c1`, `c2`, `c3` indiquent les chambres vers lesquelles chacune des portes mène. Une valeur de -1 pour une des variables `c1`, `c2` ou `c3` indique que la porte correspondante est condamnée.

Définir un appartement composé d'un vecteur de 4 `ChambreT` (`ChambreT appart[4]`) et indiquer dans les champs `c1`, `c2` et `c3` des numéros de chambres dans lesquelles on peut se rendre à partir d'une chambre donnée. Initialiser les 4 chambres aux valeurs données dans le dessin ci-dessous, puis, partant de la chambre 1, répéter jusqu'à ce qu'on tape un numéro invalide:

```
écrire le nom des chambres dans lesquelles on peut aller
demander un numéro
aller dans la chambre correspondant au numéro
```

Programme 96

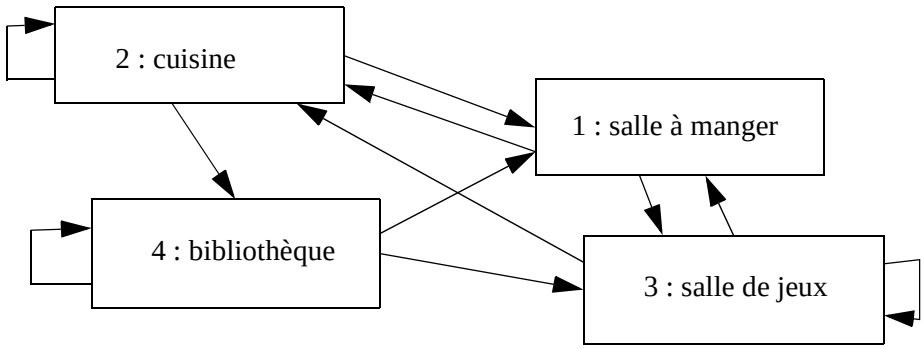


FIGURE 36: Schéma d'appartement

Labyrinthe avec pointeurs. Dans l'exercice précédent, on a défini un tableau contenant 4 chambres. Si l'on ne connaît pas, avant d'exécuter le programme, le nombre de chambres de la maison, on risque comme ci-dessus de perdre beaucoup de place en déclarant des tableaux trop grands. Cette situation est si fréquente qu'on a prévu le concept du pointeur pour utiliser la mémoire de façon optimale. Plutôt que d'avoir un index qui pointe sur un tableau, on utilise un pointeur qui est en fait une espèce d'index pointant directement sur la mémoire de l'ordinateur.

Nous allons remplacer les index par des pointeurs. Dans le premier exemple, nous définissons un pointeur pour chaque chambre. La structure avec pointeurs s'écrit comme suit:

```
typedef struct ChambreT
{
  ChambreT *c1, *c2, *c3 ;
  char nom [32] ;
} ChambreT;

ChambreT *chambres[4];

chambres[0] = (ChambreT *) malloc(sizeof(ChambreT));
```

Programme 97

Dans la première structure (Prog. 95), on repère les chambres au moyen d'entiers et dans le deuxième (Prog. 97) au moyen de pointeurs de chambre et plutôt que de mettre les chambres dans un tableau, on met seulement les pointeurs de chambre dans le tableau, et on crée les chambres en utilisant l'instruction `malloc`.

Modifier le programme du labyrinthe en utilisant des pointeurs plutôt que des indices, selon ce qui a été expliqué ci-dessus.