Paradigme Impératif et Programmation en C

Lucas Létocart et Sophie Toulouse
LIPN - UMR CNRS 7030
Institut Galilée, Université Paris 13
99 av. Jean-Baptiste Clément
93430 Villetaneuse - FRANCE
{lucas.letocart,sophie.toulouse}@lipn.univ-paris13.fr

30 Septembre 2010

Abstract

Cours accéléré de C : le but est d'y présenter les mots et concepts clés qui permettront un apprentissage plus rapide à partir d'un VRAI manuel.

Contents

1 Les premiers pas en C

1.1 La compilation et l'exécution

Tout ce qui se réfère à la compilation et à l'exécution de vos programmes en C sera expliqué au chapitre 9 et pendant les séances de TP.

1.2 Éléments de syntaxe

1.2.1 Caractères utilisés dans un programme C

On dispose en C de 92 caractères visualisables :

- Les lettres majuscules et minuscules
- Les dix chiffres décimaux
- Les 30 caractères spéciaux :

!	"	#	%	&	'	(	)	*	+
,	-	.	/	:	;	<	=	>	?
[	\	]	^	_	{	|	}	~	blanc

Ces caractères ont tous un sens syntaxique précis en C. Toutefois dans une chaîne, on peut encore placer "@" "`"(accent grave) et "$", si on utilise la codification ASCII, qui est la suivante :

32	33 !	34 "	35 #	36 $	37 %
38 &	39 '	40 (	41)	42 *	43 +
44 ,	45 -	46 .	47 /	48 0	49 1
50 2	51 3	52 4	53 5	54 6	55 7
56 8	57 9	58 :	59 ;	60 <	61 =
62 >	63 ?	64 @	65 A	66 B	67 C
68 D	69 E	70 F	71 G	72 H	73 I
74 J	75 K	76 L	77 M	78 N	79 O
80 P	81 Q	82 R	83 S	84 T	85 U
86 V	87 W	88 X	89 Y	90 Z	91 [
92 \	93]	94 ^	95 _	96 ‘	97 a
98 b	99 c	100 d	101 e	102 f	103 g
104 h	105 i	106 j	107 k	108 l	109 m
110 n	111 o	112 p	113 q	114 r	115 s
116 t	117 u	118 v	119 w	120 x	121 y
122 z	123 {	124 |	125 }	126 ~	

A ces caractères visualisables s'ajoutent des caractères non graphiques que l'on introduit à l'aide d'une barre de fraction inversée ou anti-slash (backslash) : \
Ces caractères non graphiques qui sont surtout utilisés dans les formats d'éditions sont :

retour arrière (backspace)	\b
saut de page (form feed)	\f
nouvelle ligne (new-line)	\n
retour chariot (carriage return)	\r
tabulation horizontale (horizontal tab)	\t

1.2.2 Exemple de fichier source

```
//mon premier programme C
#include<stdio.h>
void main(void)
{
    /* la fonction printf écrit une chaîne de caractères à l'ï¿½cran */
    printf("Hello, world !  !  !\n");
}
```

1.2.3 Instruction

Un programme n'est qu'une succession d'instructions ; une instruction peut être
- *simple* : elle se termine alors toujours par un point virgule "**;**" ;
- composée d'intructions simples, on parle alors de *bloc* : ces instructions sont entourées de "**{**" "**}**".

1.2.4 Commentaire

Deux marques de commentaire sont possibles :

/* texte_a_commenter (je peux passer à la ligne si je le souhaite)*/	//texte à commenter (chaque nouvelle ligne nécessite //une nouvelle marque de commentaire)

Remarque : la marque de commentaire "//" est en réalité issue du C++ et non du C ANSI ; néanmoins, elle est acceptée par les principaux compilateurs actuels (dont **gcc**) et de fait, on se permet de l'utiliser (pour plus d'informations, *cf.* paragraphe 3.6.4).

1.2.5 Variables et fonctions

Tout ce qu'on manipule en C sont des *variables* et des *fonctions*.
Une *variable* est d'un type précis et l'on peut :
- lui affecter toute valeur que l'on veut qui soit permise par son type ;
- tester sa valeur.

Une *fonction* est un ensemble d'instructions regroupées en vue d'un traitement particulier et auquel on peut faire appel à sa guise. Une fonction est caractérisée par :
- le type de donnée renvoyé ;
- la liste des types des arguments qu'elle prend en entrée.

La fonction "main" est une fonction particulière : elle est le *point d'entrée de tout programme* (C, mais pas seulement) ; de fait, elle est obligatoire (un programme peut d'ailleurs s'y réduire) et sa syntaxe n'est pas tout à fait libre ; pour en savoir plus sur la fonction **main**, se rendre au paragraphe 9.3.

Remarque : à proprement parler, il n'y a que des fonctions en C ; néanmoins, le C permet d'écrire des procédures sous la forme de fonctions qui renvoient le type particulier "void", qui signifie ... *rien*.

1.2.6 Types de base

type	*exemple*	*description*
char	char c;	caractère
int	int i;	entier relatif
float	float f;	réel
int*	int* pi;	pointeur sur une donnée de type entier
int[]	int t[];	tableau d'entiers

1.2.7 Déclarer une variable

Pour déclarer une variable, il suffit d'indiquer son *type* et son *nom* :

```
type nomVariable;
```

On peut de plus déclarer plusieurs variables au cours d'une même instruction, on peut également initialiser la valeur d'une variable au moment de sa déclaration :

```
int i; float x, y = 0; char c1 = 'a', c2 = 'b', c3 = 'c';
```

Enfin, on peut déclarer des *pointeurs* sur un type donné, des *tableaux* d'éléments d'un type donné :

```
int t[5]; //t est un tableau de 5 entiers
int* p; //p contiendra l'adresse d'une variable entière
/*Le caractère c et le pointeur s sont initialisés à la valeur nulle de leur type*/
char c = '\0', *s = NULL;
```

1.2.8 Et... concrètement ?

Lorsque le système lit `char c`, il réserve automatiquement en mémoire la place nécessaire et suffisante au stockage d'un caractère, à une adresse donnée. Cette adresse est accessible par `&c`. De la même façon, lorsque le système lit `char* pc`, il réserve en mémoire la place nécessaire et suffisante au stockage d'un pointeur, c'est-à-dire à celui d'une adresse, qui n'est autre qu'une valeur entière. L'adresse de pc est elle-même donnée par `&pc`.

1.2.9 Opérateurs

Mathématiques

+	addition	-	soustraction		
*	multiplication	/	division	%	modulo

Attention ! ! !

(1) Les opérateurs suivent certaines lois de priorité, notamment : " * " est prioritaire sur " + " ; mais de toute faï¿½on, utiliser le parenthésage !

(2) Le résultat d'une division dépend du type de ses arguments (division entière, réelle).

De comparaison

==	égal	!=	différent
<	inférieur strict	>	supérieur strict
<=	inférieur ou égal	>=	supérieur ou égal

Logiques

&&	et	||	ou	!	non

D'affectation

`var = expr`	affecte la valeur renvoyée par l'expression **expr** à la variable **var**
`var++`	incrémente la variable var : équivalent à {`var = var+1`}
`var--`	décrémente la variable var : équivalent à {`var = var-1`}
`++var`	incrémente la variable var : équivalent à {`var = var+1`}
`--var`	décrémente la variable var : équivalent à {`var = var-1`}

Remarque : les instructions `var++` et `var--` renvoient la valeur *initiale* de `var`, tandis que les intructions `++var` et `--var` renvoient la valeur *finale* de `var`.

```
int i = 0, j = 1;
i = j = 2;          //i et j valent 2
i = j++;            //i vaut 2, j vaut 3
i = ++j;            //i vaut 4, j vaut 4
```

1.3 Ce qu'il faut retenir

Les mots-clefs

Dans l'ouvrage de *Kernighan & Ritchie*, de 1978, il est présenté en tout 28 mots-clefs :

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

En dehors de *entry* qui n'est même pas défini dans l'ouvrage de référence, tous ces mots-clefs sont pris en compte par l'ensemble des compilateurs actuels.

Depuis la publication de l'ouvrage de *Kernighan & Ritchie*, en 1978, les compilateurs ont adopté des mots-clefs supplémentaires :

void enum signed const volatile

Le ";" à l'honneur

Toute instruction simple doit se terminer par un point-virgule.

Comparaison vs. affectation

Le "=" seul est une affectation ! ! ! Notamment, `while(cond = 1)` produit une boucle infinie (en plus d'affecter cond à 1), `if(i = 0)` sera toujours faux ET affecte 0 à `i`, `if (i = 2)` sera toujours vrai ET affecte 2 à `i` etc...

2 Structure d'un programme

2.1 Langage compilé

Le C est un langage *compilé*, c'est-à-dire qu'entre le programme source et l'exécutable, il faut passer par une phase de compilation qui traduit le fichier source en un fichier exécutable compréhensible par le système : le système n'est pas capable d'interprétrer directement les instructions du fichier source (contrairement au langage *shell* sous Unix par exemple qui interprète les instructions à la volée, sans passer par cette phase

de “traduction”).

2.2 Directives de précompilation

En tête de programme on place les directives de précompilation : ce sont des instructions particulières qui sont interprétées en premier lieu par le préprocesseur, avant de passer à l'étape de compilation proprement dite. Le compilateur identifie ces directives par le caractère “#” qui les précède systématiquement.

2.2.1 Directive #include

`#include<stdio.h>`
recherche le fichier “stdio.h” dans le répertoire standard
`#include "../include/pile.h"`
recherche le fichier “pile.h” dans un répertoire **include** “frère” du répertoire du fichier source
`#include "/users/Inge/toto/progC/include/oracle.h"`
recherche le fichier “oracle.h” dans le répertoire spécifié “/users/Inge/toto/progC/include”

C'est la directive la plus importante car elle permet à notre programme d'utiliser des éléments de programmation qui sont définis dans d'autres fichiers. De tels éléments sont rangés dans des librairies ou dans des fichiers appelés “*fichiers à entête*”, dont l'extension est “.h”. Ces fichiers définissent les éléments nécessaires (définition de types, constantes etc.) et le mode d'emploi (fonctions disponibles) d'une donnée particulière ou d'un type de traitement particulier. Par exemple, il peut s'agir d'un type abstrait de donnée défini par l'utilisateur (pile.h défini par le programmeur), du regroupement de fonctions pour la gestion d'une base de données (oracle.h défini par le programmeur), de fonctions d'entreé-sortie (stdio.h livrée en standard), de fonctions mathématiques (math.h livrée en standard) etc...

Les bibliothèques à connaître :

stdio.h	fonctions de gestion des entrèes-sorties
stdlib.h	fonctions d'allocation de mémoire, de conversion de chaînes, aléatoire
math.h	fonctions mathématiques
ctype.h	fonctions sur les caractères
string.h	fonctions de manipulation des chaînes de caractères
setjmp.h	fonctions de traitement des erreurs
signal.h	fonctions de gestion des évènements
time.h	fonctions du temps d'exécution

2.2.2 Directives #define, #undefine

#define permet de définir des constantes et même des fonctions nommées “*macro*”. Une macro est une fonction qui est interprétée “*en direct*” à chaque fois qu'elle apparaît dans un texte ; exemples :

#define N 5	à chaque fois que le pré-compilateur va lire **N**, il le remplacera par **5** ;
#define MULT(X,Y) (x)*(y)	à chaque fois que le compilateur va lire **MULT(arg1,arg2)**, il le remplacera par **(arg1)*(arg2)** ;
#undefine N	**N** ne sera plus reconnu à partir de cette instruction.

Remarque : le texte étant remplacé littérallement, attention à l'écriture des macros ! Sur l'exemple fourni, si l'on avait défini “#define MULT(X,Y) x*y” au lieu de “#define MULT (X,Y) (x)*(y)”, alors

l'appel "MULT(a+b,c)" n'aurait pas donné le résultat escompté "(a+b)*c", mais "a+b*c" (soit "a+(b*c)").

2.2.3 Directives #ifdef, #ifndef, #else, #elif, #endif

Les instructions **#ifdef** et **#ifndef** permettent de tester qu'un label donné correspond bien à quelque chose de défini. L'ensemble des directives **#ifdef, #ifndef, #else, #elif, #endif** permet alors de placer des instructions (qui peuvent elle-mêmes être des directives de précompilation) conditionnelles.

2.3 Déclarations

On peut dans cette partie :
- définir des types personnalisés ;
- déclarer des variables globales au fichier (dont des variables constantes) ;
- déclarer des fonctions (= en donner le mode d'emploi, ce que l'on désigne usuellement par *prototype*, *signature* ou *entête*).

```
typedef int entier;
const entier t_x = 1;
const entier t_y = 1;
//Le [;] marque la fin de l'instruction déclarative
entier Max(entier a, entier b);
//Lors de la phase déclarative, le nom des arguments est optionnel dans les fonctions
entier Min(entier, entier);
entier Moy(entier a, entier);
void translate(entier* const a, entier* const b);
//On ne spécifie la taille d'un tableau qu'à sa déclaration
//(et non quand il est argument d'une fonction)
void plus1(entier t[], int taille);
void permute1(entier* t, int taille);
```

Une fois une variable ou une fonction déclarée, cette variable ou cette fonction est connue et l'on peut s'y référer. Ainsi, une fonction non encore définie peut malgré tout être appelée, du moment qu'elle a été déclarée.

```
int i = 1;
int Max(int a, int b);
void main(void)
{
    int k = 2;
    //Ok :  la variable i et la fonction Max sont connues
    printf("Max(i,k) = %d\n", Max(i,k));
    //Ko :  la fonction Min n'est pas connue (en fait on peut mais c'est mal)
    printf("Min(i,k) = %d\n", Min(i,k));
    //Ko :  la variable j n'est pas connue
    printf("Max(j,k) = %d\n", Max(j,k));
    int j = 3;
```

```
    return;
}
int Min(int a, int b)
{
    return (Max(a,b) == a) ?  b :  a; //Ok :  la fonction Max est connue
}
int Max(int a, int b)
{
    return (a >= b) ?  a :  b;
}
```

2.4 Définitions

Le rôle de cette partie est celui de la *définition* des fonctions. Ici, on ne se contente donc pas d'écrire l'entête de la fonction, mais on en décrit aussi le corps. Notons que la définiton d'une fonction en fait par le même temps la *déclaration*.

Dans le corps des fonctions également, on commence par la partie déclarative. Quelque soit le langage utilisé, on essaie d'organiser le corps d'une fonction de la façon suivante :

1	-	déclaration des variables ;
2	-	traitements ;
3	-	retour à l'appelant.

2.4.1 Déclaration des variables

En C, on peut déclarer des variables n'importe où dans le corps d'une fonction (du moins, du moment que la variable est déclarée AVANT son utilisation, dans le même bloc ou dans un bloc de niveau supérieur). Il faut arbitrer le moment où l'on va définir une variable selon les éléments suivants :
- pour la lisibilité et la pérennité du code, il est préférable de regrouper les déclarations à un seul endroit, en début de fonction ;
- pour l'économie de mémoire, il est préférable de ne déclarer une variable qu'au moment où l'on en a besoin.

2.4.2 Retour à l'appelant

En C, on peut sortir en plusieurs endroits de la fonction (utilisation multiple de l'instruction **return**). Néanmoins, pour la lisibilité et la pérennité du code, il est préférable de n'avoir qu'un point de sortie, ne serait-ce que pour centraliser les traitements de sortie de fonction du type gestion de la mémoire, de l'erreur ou encore d'une connexion base de données.

2.4.3 Exemples

```
//Lors de la phase de définition, le nom des arguments devient obligatoire
//dans l'entête de la fonction...  sinon, comment s'y référer ? ! ?
int Max(int a, int b)
{
    return (a >= b) ?  a :  b;
}
//Un seul point de sortie dans la fonction
```

```
int Min(int a, int b)
{
    int res;
    if (a <= b)
       res = a;
    else
       res = b;
    return res;
}
//Attention :  dans la phase de définition, ne pas mettre de [;] à la fin de l'entête ! ! !
int Moy(int* const a, int* const b)
{
    return (a+b)/2;
}
//Si le type retourné est void (la fonction ne renvoie rien), l'instruction "return" en fin
//de fonction est superflue :  on sort de toute faï¿½on de la fonction sur l'accolade fermante.
void translate(int* a, int* b)
{
    *a = *a + t_x;
    *b = *b + t_y;
    return;
}
void plus1(int t[], int taille)
{
    int i = 0;
    for (i = 0 ; i < taille ; i++)
       t[i] = t[i]+1;
    return;
}
void permute1(int* t, int taille)
{
    int i = 0;
    int element_tamp;
    if (!  taille)
       return;
    element_tamp = *t;
    for (i = 0 ; i < taille-1 ; i++)
       *(t+i) = *(t+i+1);
    *(t+taille - 1) = element_tamp;
    return;
}
```

2.5 Bloc et portée

Une variable comme une fonction est connue depuis sa définition jusqu'à la fin de son bloc de définition.

```
//i n'est pas connue, j n'est pas connue
```

```
{
    ...
    int i; //i est connue, j n'est pas connue
    ...
    {
       ...
       int j; //i et j sont connues
    }
    ...  // i est connue, j n'est pas connue (j n'existe plus)
}
//i et j n'existent plus
```

Attention ! ! ! - Une variable peut en cacher une autre : on a le droit dans un bloc de définir une variable en utilisant le nom d'une variable d'un bloc parent.

```
int i = 1; //i vaut 1
{
    ...
    int i = 2; //i vaut 2
    ...
    {
       ...
       int i = 3; //i vaut 3
    }
    ...  // i vaut 2
}
//i vaut 1
```

En réalité, dans cet exemple, ce sont trois variables différentes que l'on a définies. La première n'est momentanément plus accessible car on définit la deuxième sous le même nom, mais elle redevient accessible lorsque l'on sort du bloc dans lequel la deuxième variable est définie.

2.6 Ce qu'il faut retenir

Déclaration vs. définition
Tout doit être *déclaré AVANT* utilisation, mais peut être *défini APRÈS*.
Portée
Il est autorisé de définir des objets de même nom → pour la lisibilité du code, attention à ne pas le faire.

3 Structure de contrôle

3.1 Présentation

La lecture du code est linéaire : le compilateur passe systématiquement d'une intruction à l'instruction suivante en lisant le code ligne par ligne. Or, on peut vouloir conditionner une instruction en fonction de la valeur de certaines variables, passer outre toute une partie du code lorsque l'on se trouve en erreur, ou

encore répéter une instruction sans pour autant avoir à l'écrire 100 fois : c'est là le rôle des structures de contrôle.

3.2 Discerner le vrai du faux

En C, c'est simple :

- 0 est faux ;
- tout ce qui n'est pas faux est vrai *c'est-à-dire* : est vrai tout ce qui est non nul.

La valeur nulle est :

- 0 pour une donnée de type numérique (dont fait partie le type char, si on lit le code ASCII de la donnée) ;
- '\0' pour une donnée de type char si on lit la donnée comme un caractère ;
- NULL pour un pointeur (NULL est une constante entière de valeur 0).

3.3 Instruction conditionnelle

3.3.1 Instruction if (expr) instr

L'instruction **instr** n'aura lieu que si l'experssion **expr** est évaluée à vrai ; l'instruction **instr** peut être simple ou composée.

```
if (!  (a % 2))
{
    printf("%d est pair\n" , a); //"\n" est le caractère de passage à la ligne
}
```

Remarque : il est vivement conseillé de mettre une instruction unique malgré tout entre accolades. En effet, cela rend le code encore plus lisible et surtout, cela évite les erreurs en cas d'ajout ultérieur d'une instruction.

3.3.2 Instruction if (expr) $\mathbf{instr}_1$ else $\mathbf{instr}_2$

Si l'expression **expr** est évaluée à vrai, l'instruction $\mathbf{instr}_1$ aura lieu ; sinon, c'est l'instruction $\mathbf{instr}_2$ que l'on exécute. Les instructions $\mathbf{instr}_1$ et $\mathbf{instr}_2$ peuvent être simples ou composées.

```
if (!  (a % 2))
{
    printf("%d est pair\n", a);
}
else
{
    printf("%d est impair\n", a);
}
```

Cas particulier : succession de **if - else** dans le cas d'un choix multiple. Dans ce cas précis, on présente différemment le code... mais il ne s'agit que de présentation ! On écrit la même chose, simplement il n'est plus préconisé de mettre systématiquement les accolades ni d'indenter.

```
//écriture standard
if (a % 3 == 0)
```

```
{
    printf("%d est un multiple de 3\n", a);
}
else
{
    if (a % 3 == 1)
    {
       printf("%d-1 est un multiple de 3\n", a);
    }
    else
    {
       if (a % 3 == 2)
       {
          printf("%d-2 est un multiple de 3\n", a);
       }
    }
}

//écriture simplifiée
if (a % 3 == 0)
{
    printf("%d est un multiple de 3\n", a);
}
else if (a % 3 == 1)
{
    printf("%d-1 est un multiple de 3\n", a);
}
else if (a % 3 == 2)
{
    printf("%d-2 est un multiple de 3\n", a);
}
```

Attention ! ! ! - Une instruction **else** se rapporte à la dernière instruction **if**, hors bloc **{}** et hors groupe **if + else**.

3.3.3 Instruction switch - case

Permet de tester la valeur d'une expression de type entier ou caractère.

```
switch (expr)
{
    case val^1:  instr_1^1 ; instr_2^1 ; ... instr_{n_1}^1 ; [break;]
    case val^2:  instr_1^2 ; instr_2^2 ; ... instr_{n_2}^2 ; [break;]
    .
    .
    .
    case val^m:  instr_1^m ; instr_2^m ; ... instr_{n_m}^m ; [break;]
```

```
    default:  instr_1 ; instr_2 ; ... instr_n;
}
```

L'instruction **break** (optionnelle) permet de sortir de l'instruction **switch** ; la valeur **default** (également optionnelle) est toujours vraie et permet de traiter les cas non résolus par les cas précédents.

Remarque : la valeur **default** peut servir de traitement commun à tous les cas ; il est néanmoins préférable d'extraire de tels traitements de l'instruction switch et de gérer cette valeur comme étant le cas "autre" (notamment, gestion de l'erreur si valeur inattendue) ; cela suppose que l'on conclut chaque cas avec l'instruction **break**.

3.3.4 L'opérateur ternaire "(expr) ? expr$_1$: expr$_2$;"

Renvoie **expr**$_1$ si **expr** est vraie, **expr**$_2$ si **expr** est fausse. Par exemple, l'instruction :

```
maxAB = (a >= b) ?  a :  b;
```

est équivalente à l'instruction conditionnelle :

```
if (a >= b)
    maxAB = a;
else
    maxAB = b;
```

3.4 Boucles

3.4.1 Instruction while (expr) instr

Tant que l'expression **expr** est vraie, on procède à l'instruction **instr**.

```
void remplirTab(char t[], int taille)
{
    //_________________
    // déclarations
    char c;
    int i;
    //_________________
    // traitements
    i = 0;
    while( ((c = getchar()) != EOF ) && (i < taille - 1) )
    {
       t[i] = c;
       i++;
    }
    t[i] = '\0';
}
```

3.4.2 Instruction do instr while (expr)

On procède à l'instruction **instr** jusqu'à ce que l'expression **expr** soit fausse.

```
int recherche(int t[], int e, int taille)
{
```

```
    //_________________
    // déclarations
    int i;
    int bTrouve = 0;
    //_________________
    // traitements
    if (taille)
    {
       i = 0;
       do
       {
          bTrouve = (t[i] == e);
          i++;
       }
       while( !  bTrouve && (i <= taille) );
    }
    return bTrouve;
}
```

3.4.3 Instruction for (**expr**$_1$; **eprx**$_2$; **expr**$_3$) **instr**

expr$_1$ est une instruction d'initialisation (ex. : i = 0), **expr**$_2$ est la condition à vérifier pour procéder à l'instruction **instr**, **expr**$_3$ est une instruction exécutée en fin d'itération (ex. : **i**++).

```
void affiche(int t[], int taille)
{
    //_________________
    // déclarations
    int i;
    //_________________
    // traitements
    printf("\n")
    for (i = 0 ; i < taille ; i++)
    {
       printf("%d\t", t[i]); //"\t" est le caractère de tabulation
    }
    return;
}
void affiche(int t[], int taille)
{
    //_________________
    // déclarations
    int i;
    //_________________
    // traitements
    printf("\n")
    //écriture compacte :  instr est contenue dans expr3
```

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

Remarque : cette dernière écriture est correcte car **i++** renvoie la valeur initiale de **i** ; si l'on avait écrit "`printf("%d\t", t[++i])`", alors **t** aurait été parcouru de 1 à `taille` (`t[taille]` n'existe pas ! ! !) et non de 0 à `taille-1` comme il se doit.

```
int sommePair(int t[], int taille)
{
    //_________________
    // déclarations
    int i;
    int res = 0;
    //_________________
    // traitements
    for (i = 0 ; i < taille ; i = i+2)
       res = res + t[i]
    return res;
}
```

3.5 Branchements

nom	*dans*	*sort de*	*se rend à*
continue;	une boucle	l'itération	l'itération suivante
break;	une boucle/un switch	la boucle/le switch	l'instruction suivante de la boucle/du switch
return;	une fonction	la fonction	le point d'appel dans la fonction appelante
goto;	une fonction	nulle part	l'instruction qui suit le label indiqué (peut être n'importe où dans la fonction courante)

```
//continue;
int sommePair(int t[], int taille)
{
    //_________________
    // déclarations
    int i;
    int res = 0;
    //_________________
    // traitements
    for (i = 0 ; i < taille ; i++)
    {
       //si l'indice est impair, on passe directement à l'itération suivante
       if (i%2)
          continue;
       res = res + t[i];
    }
    return res;
```

```
}

//break;
int recherche(int t[], int e, int taille)
{
    //_________________
    // déclarations
    int i;
    int bTrouve = 0;
    //_________________
    // traitements
       for ( i = 0 ; i < taille ; i++)
       {
          //dès que l'élément recherché est trouvé, on sort de la boucle
          if ( bTrouve = (t[i] == e) )
             break;
       }
    return bTrouve;
}

//return;
int recherche(int t[], int e, int taille)
{
    //_________________
    // déclarations
    int i;
    int bTrouve = 0;
    //_________________
    // traitements
    // si le tableau est vide, ce n'est pas la peine de poursuivre
    if (!  taille)
       return bTrouve;
    i = 0;
    do
    {
       bTrouve = (t[i] == e);
       i++;
    }
    while( !  bTrouve && (i <= taille) );
    return bTrouve;
}

//goto;
void main (void)
{
```

```
    //_________________
    // déclarations
    int taille = 0;
    int* t = NULL; // TOUJOURS initialiser un pointeur à NULL
    //_________________
    // traitements
    while (!  taille)
    {
       if (scanf("%d", &taille))
          taille = (taille != 0 ?  taille :  0);
    }
    t = (int*)malloc(taille * sizeof(int));
    ...
    goto FIN;
    ...
FIN:
    if (t != NULL)
    {
       free(t);
       t = NULL;
    }
    return;
}
```

3.6 Ce qu'il faut retenir : organisation du code

De manière générale : si le langage C offre une grande souplesse dans sa syntaxe, toujours penser avant tout à la lisibilité du code ($\rightarrow$ commenter, ordonner, organiser, présenter le code de sorte à ce qu'il soit d'une prise en main facile de la part d'un programmeur, mais aussi de façon à limiter le risque d'erreur et d'en faciliter la correction). Notamment : éviter les écritures compactes, ne pas hésiter à parenthéser, à mettre des accolades pour une instruction simple dans le cadre de structures de contrôle, éviter les labels etc...

3.6.1 LA règle de présentation

Afin de rendre le code plus lisible : à l'ouverture de chaque bloc, indenter le code d'un cran supplémentaire.

```
int sommePair(int t[], int taille)
{
--->//_________________
--->// déclarations
--->int i;
--->int res = 0;

--->//_________________
--->// traitements
--->for (i = 0 ; i < taille ; i++)
```

```
--->{
--->--->if (t[i]%2 == 0)
--->--->{
--->--->--->res = res + t[i]
--->--->}
--->}
--->return res;
}
```

3.6.2 Points de sortie multiples

Éviter de multiplier les points de sortie et privilégier une unique instruction **return** par fonction ! Pour ce faire : utiliser un label de sortie de fonction. Un tel label permet de :
- centraliser la sortie (on n'a pas à chercher les différents points de sortie possible dans la fonction) ;
- centraliser les traitements de sortie de fonction, tels que la libération de l'espace mémoire ou la gestion de l'erreur (transaction base de données en cours, lecture d'un fichier d'entrée etc...).

3.6.3 Labels

Éviter à tout prix les labels, sauf éventuellement un label de sortie de fonction

3.6.4 Commentaire

Commenter systématiquement le code. Notamment : expliquer les différents cas d'une expression conditionnelle ; dater, signer et motiver toute modification etc.. De plus, préférer la marque de commentaire "//", bien qu'elle ne soit pas définie dans la norme ANSI. En effet, la marque "// ..." présente l'avantage de pouvoir être incluse à l'intérieur de commentaires "/* ... */", tandis qu'on ne peut inclure de commentaire "/* ... */ " à l'intérieur d'un autre commentaire "/* ... */" (voir les exemples ci-dessous) : pour développer, il est donc préférable d'utiliser systématiquement "// ..." et de réserver la marque "/* ... */ " au test et au deboguage.

/* Ceci est un commentaire //cela en est un autre CORRECT*/	/* Ceci est un commentaire /*cela en est un autre*/ INCORRECT*/

3.6.5 Convention de nommage

Pour les fonctions : s'imposer une convention de nommage qui rende le nom explicite (à lire le nom de la fonction, je sais ce qu'elle fait, éventuellement ce qu'elle renvoie) et surtout, le nommage des fonctions à l'intérieur d'un même programme doit être homogène. Par exemple : nommer "**est**Quelquechose" toute fonction renvoyant un booléen, "**detruire**StructureDonnee" toute fonction de destruction d'un objet d'une structure de données précise.

Pour les variables : en lisant le nom d'une variable, on doit être en mesure d'en déduire le *type* et la *portée*. Il faut s'imposer ainsi une convention de nommage du type : préfixer de la lettre **g** (*resp.*, **m**) le nom d'une variable globale au programme (*resp.*, au module) ; préfixer de la lettre **c** (*resp.*, **i**, **x**, **s**, **p**, **t**) le nom d'une variable de type **char** (*resp.*, **int**, **float**, chaîne de caractères **char***, pointeur **type***, tableau **type[ ]**). Par exemple : nommer "**gti**QuelqueChose**[]**" un tableau d'entier global, "**px**QuelqueChose" un pointeur local sur un réel, "**mc**QuelqueChose" un caractére déclaré au niveau du fichier.

Remarque : il s'agit de rendre le code lisible et réutilisable, il ne faut donc pas non plus surcharger les notations ; notamment, pour les variables locales, on peut s'affranchir de certaines règles de normalisation.

4 Fonction, variable, durée de vie

4.1 Durée de vie

Généralement, une variable naît à sa définiton et meurt à la fermeture du bloc dans lequel elle est définie. Toute variable déclarée de faï¿½on statique est gérée par le système lui-même : lorsque j'écris **int i;** (*resp.*, **int t [5];**), c'est le programme qui, lui-même, se charge de réserver la place nécessaire au stockage d'un entier (*resp.*, d'une succession de 5 entiers et de l'adresse du premier). C'est encore le programme qui se charge de libérer cet espace quand la variable disparaît.

4.2 Passage par valeur

En C, les arguments d'une fonction sont tous passés par valeur. Considérons le programme suivant :

```
#include<stdio.h>
void swap(int a, int b)
{
    int tamp;
    printf("swap - a = %d, b = %d\n", a, b);
    tamp = a;
    a = b;
    b = tamp;
    printf("swap - a = %d, b = %d\n", a, b);
    return;
}
void main(void)
{
    int a = 1, b = 2;
    printf("main - a = %d, b = %d\n", a, b);
    swap(a, b);
    printf("main - a = %d, b = %d\n", a, b);
    return;
}
```

Si j'exécute mon programme, je vais voir s'afficher à l'écran :
main - a = 1, b = 2
swap - a = 1, b = 2
swap - a = 2, b = 1
main - a = 1, b = 2

C'est-à-dire que les valeurs de **a** et de **b** sont bien échangées dans la fonction **swap**, mais pas dans la fonction **main**. En effet, lorsque dans **main** on fait l'appel **swap(a, b)**, le système traduit **swap (1, 2)** et à l'entrée dans la fonction **swap**, deux variables locales sont créées pour contenir les valeurs 1 et 2, qui seront détruites en sortie de fonction swap. Autrement dit, la fonction **swap** n'aura jamais connaisssance des variables **a** et **b** de la fonction **main**. Aussi, comment faire pour qu'une fonction B appelée depuis une fonction A puisse agir sur les variables connues de A ?

4.3 Passage par adresse

4.3.1 Présentation

Nombreux sont les langages de programmation qui permettent de spécifier, quand on définit une fonction, si ses arguments sont passés par valeur ou par adresse. Le passage par adresse (ou référence) donne le droit à la fonction appelée d'agir sur les paramètres de la fonction appelante ou encore, plus précisément : ce n'est pas une copie (la valeur) des variables de l'appelant qui est envoyée, mais la variable elle-même. Seulement voilà : cela n'est pas permis en C. Pour palier cela, il suffit d'envoyer comme paramètre non pas la valeur d'une variable, mais son adresse : si la fonction appelée a connaissance de l'adresse d'une variable de la fonction appelée, et que l'on peut la faire écrire à cette adresse, la modification sera persistante.

4.3.2 Illustration 1 : fonction `swap`

Pour illustration, nous reprenons l'exemple de la fonction **swap** :

```
#include<stdio.h>
// les arguments de la fonction swap sont maintenant des pointeurs constants
void swap(int* const pa, int* const pb)
{
    int tamp;
    printf("swap - a = %d, b = %d\n", *pa, *pb);
    tamp = *pa;
    *pa = *pb;
    *pb = tamp;
    printf("swap - a = %d, b = %d\n", *pa, *pb);
    return;
}
void main(void)
{
    int a = 1, b = 2;
    printf("main - a = %d, b = %d\n", a, b);
    swap(&a, &b); //j'appelle la fonction swap sur les adresses de a et de b
    printf("main - a = %d, b = %d\n", a, b);
    return;
}
```

Si j'exécute mon programme, je vais maintenant voir s'afficher à l'écran :
main - a = 1, b = 2
swap - a = 1, b = 2
swap - a = 2, b = 1
main - a = 2, b = 1

Description du déroulement du programme :
`int a = 1, b = 2;`
Un emplacement de sizeof(int) octets (supposons 2 octets) est réservé pour la variable **a** à une adresse donnée ; en cette adresse (ex. : 1358), on peut lire :

1358 : 00000000|00000001

De même pour **b** : en **&b** (ex. : 1360), 2 octets sont réservés et initialisés à :

1360 : 00000000|00000010

On appelle la nouvelle fonction **swap** en lui passant pour arguments **&a** et **&b** : on fait précisément l'appel "**swap (1358, 1360);**".

`void swap(int* const pa, int* const pb)`
Les deux arguments **pa** et **pb** de la fonction **swap** sont alors créés : les adresses sont des entiers longs (codés sur 4 octet), on réserve donc en 2 adresses mémoires (ex. : 5632 et 5636) la place de ranger deux entiers longs. En l'adresse **&pa**, on trouve donc la valeur 1358 ou encore :

5632 : 00000000|00000000|00000101|01001100

et en l'adresse **&pb**, la valeur 1360, autrement dit :

5636 : 00000000|00000000|00000101|01010000

`int tamp;... tamp = *pa;`
La variable entière **tamp** est également créée (une zone mémoire lui est réservée) et reï¿½oit la valeur ***pa** : on cherche à lire une valeur entière à l'adresse 1358 ; **tamp** prend alors la valeur 1.

`*pa = *pb;`
En l'adresse ***pa**, on veut écrire la valeur lue en l'adresse 1360. En mémoire, en l'adresse 1358, la valeur :

1358 : 00000000|00000001

est alors remplacée par la valeur :

1358 : 00000000|00000010

`*pb = tamp;`
De même, en 1360, on remplace 2 par 1.

À la fermeture de la fonction **swap**, les variables **pa**, **pb** et **tamp** sont détruites, c'est-à-dire que les espaces **&pa**, **&pb** et **&tamp** sont libérés. De retour à la fonction **main**, les espaces mémoire pour **a** et **b** sont toujours réservés : on a toujours accès aux adresses **&a** et **&b** où l'on peut lire respectivement les entiers 2 et 1.

4.3.3 Illustration 2 : fonctions `printf` et `scanf`

Supposons que l'on travaille avec un rï¿½el **x**. On peut afficher la valeur de **x** à l'écran en faisant par exemple l'appel suivant à la fonction `printf` : `printf("x = %f\n", x);`
Inversement, on peut vouloir saisir la valeur de **x** au clavier, ce qui permet l'appel suivant ï¿½ la fonction `scanf` : `scanf("%f", &x);`

La fonction `printf` prend une valeur en argument tandis que la fonction `scanf` prend une adresse : en effet, autant `printf` n'a pas besoin de connaître plus que la valeur de la variable **x** pour l'afficher, autant `scanf` doit en connaître l'adresse pour que l'affectation soit permanente.

4.4 Ce qu'il faut retenir

4.4.1 Valeur ou adresse ?

Comment choisir, quand on écrit une fonction, de lui donner comme paramètres les valeurs ou les adresses des objets qu'on manipule ? Voici quelques éléments de décision :
1 - utiliser les valeurs lorsque la fonction n'a pas pour objet d'agir sur les variables visibles de la fonction appelante ;
2 - a contrario, utiliser les adresses lorsque la fonction agit sur les variables visibles de la fonction appelante ;
3 - utiliser les adresses lorsque les objets sont volumineux (en effet, une adresse ne prendra jamais que 4 octets tandis qu'une structure complexe peut prendre beaucoup plus !) ;
4 - quoi qu'il en soit, toujours se poser la question des conséquences de la modification des paramètres dans la fonction appelée (ces changements sont-ils permanents alors qu'on ne le souhaite pas, sont-ils temporaires alors qu'on ne le souhaite pas).

4.4.2 Constant ou non constant ?

Il est préférable, lorsque l'on passe l'adresse de variables, de déclarer les paramètres comme pointeurs constants. En effet, d'une part l'adresse d'un élément est une donnée intrinsèquement constante, d'autre part en ne déclarant pas ce pointeur comme constant, on risquerait d'aller modifier une valeur en une adresse qui ne concerne pas la fonction que l'on est en train d'écrire.

```
int* p = (int*)malloc(sizeof(int));
void swap(int* pa, int* pb)
{
    int tamp;
    pa = p;
    printf("swap - a = %d, b = %d\n", *pa, *pb);
//c'est *p et *pb que l'on échange, alors que l'on n'est peut-être pas sensé écrire en p
    tamp = *pa;
    *pa = *pb;
    *pb = tamp;
    printf("swap - a = %d, b = %d\n", *pa, *pb);
    return;
}
```

Néanmoins, une telle discipline n'a de sens que si le but est de manipuler les valeurs lisibles en ces adresses et non les adresses elles-mêmes.

5 Gestion de la mémoire

5.1 Démystifier les pointeurs

Un pointeur = une adresse ! ! ! La mémoire est une succession de bits (un bit vaut soit 0, soit 1) et ces bits sont organisés en octets (un octet est une suite de 8 bits). Selon les systèmes, les bits sont lus par paquet de 8, de 16, de 32, ou encore de 64, c'est-à-dire que selon les systèmes, un mot élémentaire réservera 1, 2, 4, ou 8 octets.

Parmi les informations qui sont associées à un type de donnée figure le nombre d'octets qu'une donnée de ce type va occuper. Par exemple, le type "char" ou caractère occupe 1 octet. En C, selon la précision

et l'ordre de grandeur souhaités, un réel (ou *float* pour nombre à virgule *flottante*) occupera 4, 8 ou encore 10 octets ; notamment, le type "float" répartit les 4 octets qui lui sont impartis en 3 octets pour les chiffres significatifs et un octet pour l'exposant (puissance de 10), plus précisément : 1 bit pour le signe du nombre, 31 bits pour la mantisse et 8 bits pour l'exposant (dont un bit pour le signe de l'exposant).

Un pointeur, quel que soit le type de donnée pointée, est toujours une variable de même taille (**long int** pour être précis) : que **p** pointe une donnée de type "char", "int[]", "float*" ou encore une donnée personnalisée, **p** ne contiendra jamais qu'une adresse mémoire ; or, une adresse est codée par un "long int" ou "entier long", soit sur 4 octets.

Alors, pourquoi typer les pointeurs ? Le typage des pointeurs permet de lire correctement la zone pointée ; autrement dit, le type du pointeur permet de répondre à la question : que dois-je m'attendre à lire à une adresse donnée ⇒ combien d'octets lire et comment les interprétrer ?

Notamment, lorque l'on réserve pour un tableau **t** une zone contingüe de mémoire, il faut savoir se déplacer de case en case : si la première valeur pointée **t[0]** se trouve à l'adresse **t**, où se trouve la valeur **t[1]** ? Plus généralement, je dois savoir aller de **p** à **p+1** ; or, en C, l'incrémentation d'un pointeur se fait en fonction de la donnée pointée : **p+1** est l'adresse de **p** *plus le nombre d'octets* nécessaires au codage de la donnée pointée par **p**. Concrètement : si **p** pointe un entier court, **p++** me place 2 octets plus loin.

5.2 Ce qu'il faut retenir

Tous les pointeurs sont de même taille, c'est-à-dire de la taille nécessaire au stockage d'une adresse mémoire.

6 Types de données

6.1 Types prédéfinis

type	*dénomination*	*nombre d'octets* *fonction processeur*	*intervalle*
char	caractère	1	-2^7 à 2^7-1
unsigned char	caractère non signé	1	0 à 2^8-1
short int	entier court	2	-2^{15} à $2^{15}-1$
unsigned short int	entier court non signé	2	0 à $2^{16}-1$
int	entier	16 bits : 2 32 bits : 4	16 bits : -2^{15} à $2^{15}-1$ 32 bits : -2^{31} à $2^{31}-1$
unsigned int	entier non signé	16 bits : 2 32 bits : 4	proc. 16 bits : 0 à $2^{16}-1$ proc. 32 bits : 0 à $2^{32}-1$
long int	entier long	4	-2^{31} à $2^{31}-1$
unsigned long int	entier long non signé	4	0 à 2^{32}-1
float	flottant (réel)	4	3.4E-38 à 3.4E38
double	flottant double (précision)	8	1.7E-308 à 1.7E308
long double	flottant double long	10	3.4E-4932 à 3.4E4932

1 2 3

[1] Un octet est la taille minimale de toute donnée en C (le type booléen, notamment, n'existe pas).
[2] Norme ASCII : 0-31 = caratères de contrï¿½le, 48-57 = 1-9, 65-90 = A-Z, 97-122 = a-z
[3] $x\mathrm{E}(-)n == x * 10^{(-)n}$

6.2 Expression de type

Est expression de type valide :
- le nom d'un type prédéfini
  ex. : `void v;`
- le nom d'un type personnalisé défini à l'aide de l'instruction `typedef`
  ex. : `typedef int* pointeurEntier; ... pointeurEntier pE;`
- une liste énumérative, une structure, une union définie par ailleurs :
  `enum nom_enum` (*resp.*, `struct nom_struct`, `union nom_union`)
  ex. : `struct complexe{float r; float i;}; ... struct complexe strC;`
- toute expression de type valide suivie de "**[]**"
  ex. : `pointeurEntier pE [];`
- toute expression de type valide suivie de "*****"
  ex. : `struct complexe* pStrC`.

6.3 Tableaux, pointeurs

Par tableau, on entend *zone mémoire contigüe* réservée pour contenir une succession de *données d'un même type*. Une telle structure peut être définie de faï¿½on statique (utilisation d'une expression constante, définition non modifiable) ou dynamique (utilisation d'une expression non nécessairement constante, l'utilisateur a la main sur la zone réservée).

Définition d'un tableau d'entiers de taille 10

tableau statique	`int t[10];`
tableau dynamique	`int* t = NULL; t = (int*)malloc(10 * sizeof(int));`

Remarques :
(1) On ne peut utiliser des tableaux statiques que lorsque la taille (du moins, un majorant de la taille) des tableaux qui seront manipulés est connue à l'avance.
(2) La taille d'un tableau statique ne peut être définie qu'une seule fois, au moment de sa déclaration ; en revanche, la taille d'un tableau dynamique peut être (re)définie autant de fois qu'on le souhaite.

Quelle que soit la méthode de définition et de déclaration d'un tableau (*i.e.*, statique ou dynamique), on accède au $i$ème élément d'un tableau (*resp.*, à l'adresse du $i$ème élément) indifféremment par **t[**i**]** ou ***(t+**i**)** (*resp.*, par **t+**i ou **&t[**i**]**).

6.3.1 Tableau (statique) monodimensionnel

Définition :

(1)	`type t[expr];`	ex. : `int t[7];`
(2)	`type t[expr] =` $\{arg_1, \ldots, arg_n\}$`;`	ex. : `int t[7] =` $\{1, 4, 2, 7\}$`;`
(3)	`type t[] =` $\{arg_1, \ldots, arg_n\}$`;`	ex. : `int t[] =` $\{1, 4, 2, 7\}$`;`

Les définitions (2) et (3) permettent d'initialiser le tableau au moment de sa définition : cela n'est évidemment possible que lorsque le contenu du tableau est connu à l'avance. Commentons maintenant ces trois instructions :

- `expr` doit être une expression constante ;
- (1) et (2) : un tableau de taille `expr` est créé ;
- (3) un tableau de la taille de la liste $\{arg_1, \ldots, arg_n\}$ est créé ;
- (2) et (3) : $t[0], \ldots, t[m-1]$ sont initialisés à $arg_1, \ldots, arg_m$ où $m = \min\{\texttt{expr}, n\}$;
- (2) si la liste contient moins de `expr` éléments, les éléments d'indice `expr+1` à $n-1$ sont initialisés à 0.

Attention ! ! ! - Avec la syntaxe "`int t[expr]` = $\{arg_1, \ldots, arg_n\}$", il n'y a pas de vérification quant à l'adéquation de la taille du tableau et du nombre d'éléments de la liste : aussi, si n dépasse `expr`, on ira écrire au-delà de la place réservée par **t**.

Remarque : pour utiliser un tableau (statique comme dynamique) déclaré à l'extérieur (de la fonction, du fichier), il suffit de déclarer `type t[];` (ex. : `int t[];`), sans préciser la taille.

6.3.2 Cas particulier des tableaux de caractères

Pour qu'un tableau de caractères puisse être manipulé comme une chaîne de caractères[4], il doit contenir le caractère de fin de chaîne `'\0'`. Aussi, lorsqu'on initialise le tableau en fournissant une liste de caractères et que l'on souhaite manipuler ce tableau comme une chaîne de caractères, il faut que celle-ci se termine par le caractère `'\0'`, ex. : `char t[5] = {'d', 'z', 'S', 'c', '\0'}`. De plus, une syntaxe supplémentaire est autorisée pour l'initialisation des tableaux de caractères :

```
char* s = "chaine_de_caracteres";
```

Si la chaîne de caractères (ex. : "Coucou !") est de longueur n (ex. : 8), un espace de $n+1$ caractères (ex. : 9) est réservé à l'adresse attribuée à la variable **s** et initalisé à :

```
'C', 'o', 'u', 'c', 'o', 'u', ' ', '!', '\0'
```

Ainsi, le caractère de fin de chaîne est ajouté automatiquement. Dernier élément de souplesse supplémentaire : la chaîne **s** peut être réinitialisée autant de fois qu'on le souhaite.

```
char* s = "Bonjour !";
s = "Hola !";
s = "Kalimera !";
```

Attention ! ! ! - Lorsque j'écris "`s = "Hola !";`", **s** est un tableau de taille 7 dont je peux *lire* seulement, mais *non pas modifier*, les éléments.

6.3.3 Tableau (statique) multidimensionnel

Définition :

(1) `int t[expr`$_1$`][expr`$_2$`],...,[expr`$_k$`];` ex. : `int t[4][2][3];`
où `expr`$_1$,`expr`$_2$,...,`expr`$_k$ sont des expressions constantes

L'accès à l'élément d'indice $(i_1, i_2, \ldots, i_k)$ est donné par l'appel : `t[`i_1`][`i_2`],...,[`i_k`]`. De même que pour les tableaux à une dimension, on peut initialiser un tableau multidimensionnel au moment de sa déclaration. Par exemple : `int t[2][3] =` $\{\{1,2,3\},\{4,5,6\}\}$ ou encore `int t[2][3] =` $\{1,2,3,4,5,6\}$ (le découpage peut être déduit par la connaissance de la taille de chaque dimension) ou encore : `int`

[4] C'est-à-dire : être reconnue comme tel pour l'affichage (%s), être argument des fonctions de manipulation de chaînes telles `strlen` et `strcmp` etc..

`t[][3]` = $\{1, 2, 3, 4, 5, 6\}$ (la taille de la première dimension peut être déduite de la taille des autres dimensions).

t est une matrice (2×3) :

1	2	3
4	5	6

Pour utiliser un tableau (statique comme dynamique) déclaré à l'extérieur (de la fonction, du fichier), il suffit de déclarer : `type t[][expr`$_2$`],...,[expr`$_k$`];`, par exemple : `int t[][2];`. De nouveau, on peut omettre de préciser la taille de la première dimension.

6.3.4 Pointeurs

En C, un pointeur est un nombre entier égal à l'adresse effective (adresse attribuée par l'édition de liens) en mémoire d'un objet. Bien qu'étant un nombre entier, un pointeur n'appartient pas à la famille des entiers du type `integral`. La qualité de pointeur est un type en soi, type dérivé de tous les types existants en C. Il s'agit d'un type dérivé parce qu'on ne définit pas un pointeur comme étant seulement un nombre entier ; on le définit aussi par rapport au type de l'objet qu'il est censé pointer.

Pointeurs constants et pointeurs variables La façon la plus naturelle de mettre en évidence un pointeur constant est d'introduire, dans une expression, l'adresse d'un objet de la mémoire grâce à l'opérateur d'adresse &.

Un pointeur variable est contenu dans un objet. Un objet devant contenir un pointeur variable est du type "pointeur vers un objet de type donné". On distingue les pointeurs scalaires et les tableaux de pointeurs, ces derniers seront vus plus loin. Un pointeur variable scalaire est déclaré sous la forme :

```
type * identificateur;
```

'*' est ici le symbole de déclaration d'un pointeur, "identificateur" est le nom de l'objet qui pourra contenir un pointeur et "type" est le type d'un objet vers lequel devra pointer le pointeur ainsi déclaré.

Il faut bien noter que la déclaration ci-dessus entraîne seulement la réservation d'une zone mémoire susceptible de contenir un pointeur, c'est-à-dire une adresse. Cette seule déclaration ne saurait suffire pour affecter une valeur à l'objet.

L'opérateur unaire '*' dit opérateur d'indirection permet d'atteindre la variable pointée par un pointeur.

Les pointeurs de pointeurs Un objet de type pointeur a lui-même une adresse en mémoire, on peut ainsi définir un objet de type pointeur de pointeur. Il va de soi que l'on pourra introduire des pointeurs de pointeurs de pointeurs, etc... Ces pointeurs de pointeurs seront mis naturellement en œuvre dans le cadre des tableaux multidimensionnés.

Les opérations sur les pointeurs Nous ne considérons ici que les pointeurs sur les scalaires.

Les seules opérations arithmétiques applicables aux pointeurs scalaires sont :

- L'addition à un pointeur d'une expression de type integral. Le résultat est un pointeur de même type que l'opérande pointeur.
- La soustraction à un pointeur d'une expression de type integral. Le résultat est un pointeur de même type que l'opérande pointeur.

- La différence de deux pointeurs, pointant tous deux vers des objets de même type. Le résultat est un entier.

Notons que la somme de deux pointeurs n'est pas autorisée en C car la somme de deux adresses n'est jamais définie.

Les deux opérateurs d'incrémentation (++) et de décrémentation (– –) sont applicables aux variables pointeurs.

Les tableaux de pointeurs On déclare un tableau de pointeurs monodimensionné sous la forme :

```
type * identificateur[n];
```

"type" peut être le nom d'un type quelconque, "identificateur" est le nom du tableau de pointeurs et "n" est la dimension du tableau. Il ne faut pas confondre cette déclaration de tableau de pointeurs avec la déclaration :

```
type (*identificateur)[n];
```

qui concerne un pointeur d'un tableau "identificateur" de "n" éléments de type "type".

On peut de même déclarer un tableau de pointeurs multidimensionné :

```
type * identificateur[m][n][p];
```

Signalons qu'il existe une relation de type entre un tableau de pointeurs et un pointeur de pointeur. En effet, un tableau de pointeurs monodimensionné pointant vers des objets de type donné, a le même type qu'un pointeur de pointeur vers un objet de même type. Il s'agit d'une équivalence concernant seulement le type.

6.3.5 Les pointeurs et les tableaux

Un tableau est une constante de type pointeur. L'orsqu'un nom de tableau figure dans une expression, il désigne une constante de type pointeur. La valeur de cette constante correspond à l'adresse du premier élément du tableau, c'est-à-dire &tab[0], tab étant le nom du tableau concerné. On pourra affecter cette valeur constante à une variable pointeur censée pointer elle-même vers un objet de même type que le tableau en question.

Il existe une relation entre l'opérateur d'indexation [] et l'opérateur d'indirection * :

```
expr[expi] = *(expr + expi)
```

On constate alors l'équivalence existant entre un tableau et un pointeur.

Rappelons que, lorsque l'on déclare un tableau `type t[m][n]` cela signifie qu'il sera réservé en mémoire $m * n$ objets contigus du type indiqué. lorsque l'on déclare un tableau de pointeurs `type *pt_t[m]`, il sera réservé en mémoire m objets contigus, chaque objet correspondant à un pointeur.

6.3.6 Tableaux dynamiques

On peut définir un pointeur sur un type (`type *p;`) et à partir de ce pointeur réserver autant de données du type pointé que l'on veut, en utilisant la fonction d'allocation **malloc** :

`p = (type*)malloc(`n`* sizeof(type))` où n est une variable entière.

C'est ce que l'on appelle un tableau *dynamique*. L'avantage d'une telle allocation est que, souvent, on ne connaît la taille du tableau qu'en cours de programme, cette taille pouvant dépendre d'un fichier de données lu en entrée, de données saisies par l'utilisateur etc... Cette taille peut d'ailleurs évoluer en cours de programme, aussi avons-nous la possibilité de réallouer, par la fonction "`realloc`" :

`p = (type*)realloc(p,` n'`* sizeof(type))` où n' est une variable entière.

Si n' est plus grand que n, les donnï¿½es sont prï¿½servï¿½es. Enfin, après toute allocation dynamique de mémoire, il faut lorsque la donnée ne nous intéresse plus libérer l'espace mémoire en utilisant la fonction de désallocation "`free`" :

```
if (p) {...  free(p); p = NULL;};
```

Attention ! ! !

(1) Il est préférable de ne pas abuser de la fonction "`realloc`" : en effet, ce n'est pas parce qu'un espace est *libéré* qu'il est *nettoyé*. Aussi, si l'on procède à la réallocation d'un tableau d'une taille 10 à une taille 5, les 5 derniers éléments du tableau "avant réallocation" ne seront pas écrasés tant que l'espace mémoire qu'ils occupent n'aura pas été utilisé pour autre chose (aspect sur lequel le programmeur ne prend généralement pas la main). De fait, on pourra continuer de lire des données au-delà du cinquième élément, sans se rendre compte qu'il s'agit d'une erreur et que l'on est allé trop loin (cas de bogue non systématique... les plus pernitieux !). Alors en cas de réallocation et quand cela est pertinent, penser à initialiser l'espace alloué (mettre des valeurs nulles).

(2) - TOUJOURS libérer la mémoire après allocation dynamique. Sinon, le programme aura du mal à s'exécuter correctement : il perdra du temps ï¿½ rechercher de l'espace mémoire disponible, voire il devra s'arrêter pour cause de mémoire insuffisante.

(3) Gare à ne pas libérer ce que l'on ne souhaite pas perdre ! De manière plus générale, en matière d'espace alloué, ne pas s'attacher aux variables mais à leur valeur : j'ai deux pointeurs d'entiers **t1** et **t2**, je peux allouer en **t1** à un moment donné, faire ensuite **t2** = **t1**, puis plus tard encore, faire pointer **t1** sur autre chose tandis que la zone allouée (encore accessible par **t2**) m'intéresse toujours. À ce moment là, pour lire les données initialement placées en **t1**, je dois consulter **t2** ; si je fais l'appel "`free(t1)`", je libère l'espace *éventuellement* alloué en **t1** (je ne sais pas où pointe **t1** à ce moment), mais je ne touche pas à la zone pointée par **t2**.

```
//Fonction d'affichage d'un tableau d'entiers
void affiche(int* t, int n)
{
    int i;
    for (i = 0 ; i < n ; printf("%d\t", t[i++]));
    return;
}
//Fonction de remplisage interactif d'un tableau d'entiers
void remplir(int* t, int n)
{
    int i;
    for (i = 0 ; i < n ; scanf("%dt", &t[i++]));
    return;
}
```

```
//Fonction principale - programme de manipulation d'un tableau d'entiers
void main(void)
{
    int n = 0, *t = NULL, *ct = NULL, *cct = NULL;
    while (!  n)
       if (scanf("%d", &n));
          n = (n >= 0 ?  n :  0);
    t = (int*)malloc(n * sizeof(int));
    // On a le droit de faire une affectation multiple x = z = z = t = 0, mais EVITER !
    cct = ct = t;
    remplir(t, n);
    affiche(t, n); //affiche le contenu de t
    affiche(ct, n); //affiche toujours le contenu de t = ce qui est lu en ct == t
    remplir(ct, n); //ct == t, on est en train de redéfinir le contenu de t !
    t = NULL; //t vaut NULL
    affiche(ct, n); //affiche toujours le contenu de ct
    affiche(t, n); //ERREUR : t vaut NULL ssi t ne pointe plus sur rien !  !  !
    free(t); //ERREUR : t vaut NULL ! !  !
    free(cct); //libère l'espace alloué en l'adresse cct
    free(ct); //ERREUR : l'espace sur lequel ct pointait vient tout juste d'être libéré !
    return;
}
```

Tableau de pointeurs Tableau comme pointeur se définissant sur une expression de type, on peut donc définir des tableaux de pointeurs ou des pointeurs de tableaux de la même faï¿½on que l'on peut définir des tableaux à plusieurs dimensions, des pointeurs sur des pointeurs ou des tableaux (*resp.*, des pointeurs) sur des types personnalisés.

Pour illustrer l'utilité des tableaux de pointeurs, prenons l'exemple de la représentation d'un graphe sous forme de listes d'adjacence. Un graphe est la donnée d'un ensemble $V = \{v_1, \ldots, v_n\}$ de n sommets et d'un ensemble $E \subseteq V \times V$ de m arêtes sur ces sommets. Pour représenter un graphe, plusieurs solutions sont possibles, notamment la représentation par listes d'adjacence : il s'agit d'associer à chaque sommet v la liste $\Gamma(v)$ des sommets qui lui sont adjacents. Pour ce faire, prenons un tableau de taille n pour lequel l'indice i représente le sommet v_{i+1}. En l'indice i, je m'attends alors à lire la liste des sommets adjacents à v_{i+1}, elle-même représentée par un tableau : $*(t+i)$ sera un tableau de $|\Gamma(v_{i+1})|+1$ entiers, où $*(*(t+i))$ donne le nombre $|\Gamma(v_{i+1})|$ des sommets adjacents à v_{i+1} et $*(*(t+i)+1), \ldots, *(*(t+i)+*(*(t+i)))$ est la liste d'adjacence de v_{i+1}. Par exemple, le graphe $G = (V, E)$ défini par $V = \{0, 1, 2, 3\}$ et $E = \{10, 12, 13, 03\}$ sera représenté de la façon suivante :

t :

0	1	2	3	sommet
2	3	1	2	longueur des listes d'adjacence
1	0	1	0	description des listes d'adjacence
3	2		1	
	3			

Ainsi, le graphe est représenté par un tableau de tableaux dynamiques dont voici l'implémentation des fonctions de création et de destruction :

```
//Initialisation d'un graphe
int** creerGraphe(int nbSommet)
{
   int** g = NULL;
   if (nbSommet > 0)
   {
      g = (int**)malloc(nbSommet * sizeof(int*));
      for (i = 0 ; i < nbSommet ; i++)
         *(g+i) = NULL;
   }
   return g;
}
//Initialisation de la liste d'un sommet
int* creerAdj(int nbAdj)
{
   int* a = NULL;
   if (nbAdj > 0)
   {
      a = (int*)malloc(nbAdj * sizeof(int));
      *a = nbAdj;
   }
   return a;
}
//Destruction d'un graphe
void detruireGraphe(int** g, int nbSommet)
{
   int i;
   if (g)
   {
      for (i = 0 ; i < nbSommet ; i++)
         if (g[i])
            free(g[i]);
      free(g);
   }
} //Manipulation d'un graphe
void main(void)
{
   int** g = NULL;
   int nbSommet = 0, nbAdj = 0;
   int i;
   ...
   g = creerGraphe(nbSommet);
   ...
   for (i = 0 ; i < nbSommet ; i++)
   {
```

```
      ...
      *(g+i) = creerAdj(nbAdj);
      ...
   }
   ...
   if (g)
   {
      detruireGraphe(g, nbSommet);
      g = NULL;
   }
}
```

Tableau générique Tableau générique : si l'on ne connaît pas à l'avance le type de donnée pointé par les éléments du tableau (ou que le tableau contient des données hétérogènes), on peut ne pas spécifier le type pointé en utilisant le pointeur générique **void*** :

```
void** t = (void**)malloc(n * sizeof(void*));
```

En effet, les pointeurs étant tous de même taille, la place à réserver ne dépend finalement que du nombre de pointeurs (adresses) à réserver. On peut spécifier plus tard, au moment de son affectation, le type pointé par le pointeur **t[i]** :

```
(int*)t[2] = (int*)malloc(7 * sizeof(int));
```

Il faut alors pour manipuler les éléments de **t** rappeler qu'il s'agit d'un tableau d'entiers ; par exemple, pour lire **t[2]**, il faut écrire :

```
printf("3ième élément du tableau t[2] = %d\n", (int*)(t[2])+3)
```

De mono- à multi-dimensionnel Dans le cadre statique comme dans le cadre dynamique, entre mono- ou multi-dimensionnel, il ne s'agit que d'un choix d'implémentation ; en effet, un tableau tMULTI à k dimensions de taille $(n_1, \ldots, n_k)$ est toujours représentable par un tableau tMONO à 1 dimension, en définissant :

$$\text{tMULTI}[i_1][i_2], \ldots, [i_{k-1}], [i_k] == \text{tMONO}[(n_2 * \ldots * n_k)i_1 + (n_3 * \ldots * n_k)i_2 + \ldots + n_k * i_{k-1} + i_k]$$

Par exemple, sur le tableau du paragrahe 6.3.3 : `t[1][2] == 5 == tMONO[3 * 1 + 2]`. ***Attention*** justement à limiter la profondeur des liens de type pointeur manipulés : "**" va encore, au-delà se méfier.

6.3.7 Statique vs. dynamique ?

L'allocation statique a l'avantage d'une simplicité (et donc d'une lisibilité) du code ainsi que de la centralisation des aspects d'allocation mémoire, les tailles réservées devant être incorporées dans des constantes. On ne peut cependant prendre ce parti que lorsque l'on connaît à l'avance la taille (ou du moins un majorant de la taille) des données manipulées. Aussi, quand de tels éléments sont connus, est-il préférable de choisir des définitions statiques, *tant que l'encombrement mémoire n'est pas problématique.*

6.4 Ce qu'il faut retenir

6.4.1 Allocation dynamique de mémoire

Dans un programme en C, on peut allouer et libérer dynamiquement des zones mémoires grâce aux fonctions suivantes :
malloc
calloc
free
realloc

- La fonction **malloc(n)** est une fonction pointeur vers un objet de type quelconque. Cette fonction retourne un pointeur vers un objet dont la taille en octets est égale à **n**. Le pointeur retourné pointe vers le début de la zone mémoire allouée. Ce pointeur sera adapté au type de l'objet visé et les problèmes d'alignement en mémoire seront résolus automatiquement. La zone allouée par **malloc** réside dans la partie temporaire dynamique de la mémoire. Il en est de même pour **calloc** et **realloc**. Si l'espace demandé par **malloc** ne peut être alloué ou encore si **n** est égal à zéro, la valeur retournée par cette fonction est celle du pointeur nul.

- La fonction **calloc(n_elem,n)** est analogue à **malloc** sauf que :
  La zone mémoire allouée correspond à présent à un tableau de **n_elem** éléments et chacun de ces éléments a une taille de **n** octets.
  Cette zone mémoire allouée est mise systématiquement à zéro, dès l'allocation.

- La fonction **free(pointeur)** libère dynamiquement une zone mémoire du segment temporaire dynamique préalablement allouée par **malloc**, **calloc** ou **realloc**. L'argument **pointeur** a la valeur de l'adresse d'une zone allouée au cours d'un appel précédent à l'une de ces fonctions. La fonction **free**, qui est du type **void**, ne retourne aucune valeur.

- La fonction **realloc(pointeur,n)** permet de modifier une zone mémoire temporaire dynamique déjà allouée par **malloc**, par **calloc** ou même éventuellement par **realloc**. L'argument **pointeur** a la valeur du pointeur qui a été retournée précédemment par une de ces fonctions. La nouvelle taille de la zone mémoire sera égale à **n** octets. La fonction **realloc** retourne un nouveau pointeur, **pointeur_p**, qui pointe vers le début de la zone mémoire. S'il est impossible d'allouer une nouvelle zone mémoire par **realloc**, la valeur du pointeur retourné, **pointeur_p**, est nulle. Cependant, dans ce cas, la zone déjà allouée ainsi que la valeur de **pointeur** resteront inchangées.

Contrôles élémentaires :
- *toujours* - avoir autant d'instructions **malloc** que d'instructions **free** ;
- *dans la mesure du possible* - allouer et désallouer un pointeur dans la même fonction (quitte à, dans cette fonction, faire appel à une fonction d'initialisation pour l'allocation et une fonction de destruction pour la libération).

Règles de gestion
Il revient au programmeur de gérer ses allocations mémoire ; pour se faciliter la tâche, celui-ci peut s'imposer quelques règles simples mais salvatrices de gestion, qui n'ont de sens que si elles sont appliquées *toutes* et *systématiquement* :
- *toujours* - au moment de la déclaration, initialiser un pointeur à NULL ;
- *toujours* - au moment de la libération, réinitialiser un pointeur à NULL ;
- *toujours* - tester si un pointeur est non NULL avant d'appeler la fonction **free** sur ce pointeur.

Fonction "malloc" :

- *toujours* - utiliser la fonction "sizeof", même si l'on connaît la taille du type pointé (pour la portabilité du code) ;

- soit **t** une variable, ***t** désigne implicitement le type pointé par **t** ; notamment, on peut utiliser pour l'allocation l'expression :

`t = (type)malloc(sizeof(*t))` si `type` est un type pointeur.

6.5 Les pointeurs de fonctions

Les pointeurs de fonction sont notamment utilisés pour introduire des paramètres fonction dans la liste des paramètres d'une fonction particulière. Dans ce cas, l'usage d'un pointeur d'une fonction "f" doit être assujetti à des règles d'emploi précises :

1. Il est nécessaire de déclarer la fonction "f" dans la fonction appelante sous la forme : `type f()`
2. Dans la liste d'appel, figure cette fonction sous la forme : `f`
3. Dans la fonction qui utilise "f" comme paramètre il faudra redéclarer "f", en tant que paramètre effectif, sous la forme : `type (*f)()`
4. Dans le bloc principal de la fonction qui utilise "f", "f" interviendra sous la forme : `(*f)(liste des paramètres effectifs)`
5. Enfin la corps de la fonction "f" elle-même comportera, en tête, la déclaration habituelle : `type f(liste des paramètres formels)`

6.6 Liste énumérée

Le mot réservé **enum** permet de donner des valeurs numériques à des expressions symboliques :

`enum nom_enum{label`$_1$ `[=VAL`$_1$ `], label`$_2$ `[=VAL`$_2$ `], ..., label`$_n$ `[=VAL`$_n$ `]}`

où VAL_1,VAL_2,...,VAL_n sont des entiers naturels (optionnels).

Ex. : `enum couleur{ROUGE = 2, VERT = 0, ORANGE = 1};`

À la suite de cette déclaration, `ROUGE` vaut 2, `VERT` vaut 0 et `ORANGE` vaut 1. Pour l'instant, cela revient à avoir défini 3 constantes :

```
const int ROUGE = 2, VERT = 0, ORANGE = 1;
```

En fait **enum** permet plus, puisque maintenant on peut déclarer des variables de la liste énumérée **nom_enum** (**couleur** dans l'exemple), chose que l'on ne peut pas faire avec de simples constantes :

`enum nom_enum nom_variable;` ex. : `enum couleur etat_feu;`

En quelque sorte, **enum** permet de regrouper des constantes symboliques sous un label commun. Pour simplifier la syntaxe de déclaration d'une variable d'une liste énumérée, on peut définir un alias de type par l'instruction **typedef** :

`typdef enum nom_enum nom_type;` (ex. : `typdef enum couleur COULEUR;`)

Il suffit ensuite de déclarer :

`nom_type nom_variable;` (ex. : `COULEUR etat_feu;`)

Remarques

(1) Pour le compilateur, une variable d'une liste énumérée peut prendre toute valeur entière positive ; ainsi, poursuivant l'exemple précédent, les affectations "`etat_feu = ROUGE;`", "`etat_feu = 0;`" et "`etat_feu = 10;`" sont toutes ausi correctes les unes que les autres.

(2) On peut ne pas préciser la valeur numérique des éléments de la liste ; les règles d'affectation appliquées sont alors les suivantes :

- si le label courant est le premier label de la liste, il prend la valeur **0** ;
- sinon, il prend la valeur du label précédent $+\mathbf{1}$.

Ex., si l'on définit :

```
enum arcenciel{ROUGE, ORANGE=2, JAUNE=4, VERT, BLEU, INDIGO=10, VIOLET};
```

l'instruction :

```
printf("ROUGE=%d\t VERT=%d\t BLEU=%d\t VIOLET=%d\n", ROUGE, VERT, BLEU, VIOLET);
```

affiche à l'écran :

```
ROUGE=0  VERT=5  BLEU=6  VIOLET=11
```

(3) On peut définir la liste et le type dans une seule et même instruction ; le nom de la liste est alors optionnel :

`typedef enum [nom_enum]{label`$_1$`[=VAL`$_1$`], label`$_2$`[=VAL`$_2$`],..., label`$_n$`[=VAL`$_n$`0]} nom_type`

Ex. :

```
typdef enum couleur{VERT, ORANGE, ROUGE} COULEUR; ou
typdef enum {VERT, ORANGE, ROUGE} COULEUR;
```

6.7 Structure

Les structures permettent de définir des données composées. Par exemple, lorsque l'on manipule un tableau, une donnée qui lui est intimement liée est sa taille : aussi est-il légitime de souhaiter grouper **int t[]** et **int taille** à l'intérieur d'une même entité. Autre exemple : on souhaite manipuler les nombres complexes ; or, un tel nombre est composé de sa partie réelle et de sa partie imaginaire ; de nouveau, on souhaiterait regrouper ces deux informations à l'intérieur d'une entité "complexe" ; c'est ce que permettent les structures.

```
struct nom_structure
{
    type_donnee_1   nom_donne_1;
    type_donnee_2   nom_donne_2;
    ...
    type_donnee_n   nom_donne_n;
};
```

Ex. :

```
struct complexe
{
    float  r;
    float  i;
};
```

```
struct tableau
{
    int   taille;
    int*  t;
};
```

On peut alors déclarer une variable de la structure :

`struct nom_structure nom_var;` (ex. : `struct complexe c; struct tableau tab;`)

Et accèder aux différents champs de cette variable par l'opérateur "`.`" :

`nom_var.nom_donne_1, nom_var.nom_donne_2,..., nom_var.nom_donne_n`

Ex. : `c.r = 2.1; c.r = -3.5;  tab.t = (int*)malloc(tab.taille * sizeof(int));`

On peut enfin, tout comme pour les listes énumérées, définir un alias de type :
`typedef struct nom_structure nom_type;`
Ex. : `typedef struct complexe COMPLEXE;  typedef struct tableau TABLEAU;`

L'affectation est la seule opération globale applicable aux structures.

Remarques
(1) On peut définir la structure et le type dans une seule et même instruction ; le nom de la structure est alors optionnel :
`typedef struct {float r; float i;} COMPLEXE;`

(2) Souvent, ce n'est pas un alias de la structure que l'on définit comme type, mais celui d'un *pointeur sur la structure* : `typedef struct {float r; float i;}* pCOMPLEXE;`
En effet, dans la pratique, ce sont les adresses (beaucoup plus légères !) et non les objets qui sont manipulées ; autant alors directement définir le type correspondant à ce que l'on va manipuler, c'est-ï¿½-dire les adresses des données d'une structure précise.

(3) Accès aux membres de la structure : un second opérateur, "$->$", permet d'accéder aux membres d'une structure et se réfère à l'adresse de la structure et non plus à la structure elle-même.
`&nom_var->nom_donne_1`, `&nom_var->nom_donne_2`,...,`&nom_var->nom_donne_n`
Ex1. : `(&c)->r = 2.1; (&c)->r = -3.5;`
Ex1. : `(&tab)->t = (int*)malloc((&tab)->taille * sizeof(int));`

Il s'agira alors de choisir la syntaxe la plus légère en fonction de ce que l'on manipule (*i.e.*, structure ou pointeur sur structure), en ayant connaissance des équivalences suivantes (données sur l'exemple de la structure "tableau") :

	`struct tableau* ptab;`		`struct tableau tab;`
$\Rightarrow$	`(*ptab).t == ptab->t`	$\Leftrightarrow$	`tab.t == (&tab)->t`
	`(*ptab).taille == ptab->taille`		`tab.taille == (&tab)->taille`".

(4) Définition récursive : on peut définir une structure dont un des membres est lui-même de la structure que l'on est en train de décrire. Cette syntaxe est notamment utilisée pour manipuler des listes chaînées :

```
struct maillon
{
   int element;
   struct maillon* suiv;
}
```

Un maillon n'est autre que la donnée d'une valeur (élément contenu dans le maillon) et de l'adresse du maillon suivant ; une chaîne est alors définie comme un pointeur sur son premier maillon :
`typedef struct maillon* CHAINE;`
La représentation sous forme de liste chaînée d'une liste d'éléments de type homogène est une alternative au tableau : elle offre plus de souplesse, les maillons pouvant être créés et détruits à tout moment, tandis que dans un tableau (même dynamique), il faut à un moment donné figer le nombre d'éléments.

C'est en fonction de l'utilisation que le programme fera de la liste que doit être fait le choix d'implémentation : par exemple, dans une liste chaînée de taille 10, détruire le deuxième maillon revient en gros à actualiser le pointeur du maillon qui le précède pour lui affecter l'adresse du maillon qui lui succède... tandis que dans un tableau, cela nécessite de décaler d'un cran toutes les valeurs des indices 2 à 9 ; en revanche, dans un tableau, on sait lire instantanément la valeur du $i$ème élément, tandis que la liste chaînée nécessite le parcours de tous les maillons depuis la premier maillon jusqu'au maillon recherché.

Remarque : Il existe une relation entre l'opérateur "->" pointeur de membre de structure et l'opérateur "." de membre de structure. En effet :

pointeur_de_structure->membre	est équivalent à	(*pointeur_de_structure).membre
variable_structure.membre	est équivalent à	(&variable_structure)->membre

6.8 Union

Une **union** désigne un ensemble de variables de types différents susceptibles d'occuper alternativement une même zone mémoire.

Les spécifications et règles d'utilisation relatives aux variables de types **union** et **struct** sont identiques.

La seule différence entre le type **struct** et le type **union** est que le premier désigne les différents membres des zones mémoire consécutives, tandis que le second est constitué d'une seule zone mémoire commune à tous les membres.

Si les membres d'une **union** sont de longueurs différentes, il est réservé une zone mémoire suffisante pour contenir le plus encombrant d'entre eux.

Exemple :

```
union u_var
{
   char c;
   float x;
}
typedef union u_var equivalence;
main()
{
   equivalence u;    u.c = 'E';    printf("%c\n", u.c + 1 );
   u.x = 3.8;    printf("%f\n", u.x + 1 );
}
```

Résultat :

```
F
4.800000
```

6.9 Listes chaînées

La structure la plus utilisée pour manipuler des données est le tableau, qui contrairement aux listes chaînées, est implémenté de façon native dans le langage C. Cependant dans certains cas, les tableaux ne constituent pas la meilleure solution pour stocker et manipuler les données. En effet, pour insérer un élément dans un tableau, il faut d'abord déplacer tous les éléments qui sont en amont de l'endroit où l'on souhaite effectuer l'insertion, ce qui peut prendre un temps non négligeable proportionnel à la taille du

tableau et à l'emplacement du futur élément. Il en est de même pour la suppression d'un élément ; bien sûr ceci ne s'applique pas en cas d'ajout ou de suppression en fin de tableau. Avec une liste chaînée, le temps d'insertion et de suppression d'un élément est constant quelque soit l'emplacement de celui-ci et la taille de la liste. Elles sont aussi très pratiques pour réarranger les données, cet avantage est la conséquence directe de la facilité de manipulation des éléments. Plus concrètement, une liste simplement chaînée est en fait constituée de maillons ayant la possibilité de pointer vers une donnée accessible et modifiable par l'utilisateur ainsi qu'un lien vers le maillon suivant. Une liste chaînée est une succession de maillons, dont le dernier pointe vers une adresse invalide (NULL). Bien sûr, ce n'est qu'une représentation possible, il se peut que les structures qui composent la liste ne soient pas placées dans l'ordre en mémoire et encore moins de façon contiguë. Au vu de l'utilisation des listes chaînées, il se dessine clairement quelques fonctions indispensables :

- Initialisation
- Ajout d'un élément
- Suppression d'un élément
- Accès à l'élément suivant
- Accès aux données utilisateur
- Accès au premier élément de la liste
- Accès au dernier élément de la liste
- Calcul de la taille de la liste
- Suppression de la liste entière

Le principal problème des listes simplement chaînées est l'absence de pointeur sur l'élément précédent du maillon, il est donc possible de parcourir la chaîne uniquement du début vers la fin. À la différence des listes simplement chaînées, les maillons d'une liste doublement chaînée possèdent un pointeur sur l'élément qui les précèdent. Le fait d'avoir accès à l'élément précédent va permettre de simplifier certaines fonctions :

- Il n'est plus nécessaire d'utiliser une sentinnelle pour retrouver le premier élément de la liste ce qui permet de se passer de la structure qui englobait la liste simplement chaînée.
- L'insertion d'un élément peut se faire aussi bien avant qu'après celui passé en paramètre de la fonction
- Il est maintenant possible de supprimer l'élément passé en paramètre et non plus le suivant
- Il faut une fonction qui retourne le maillon précédent

À part ces quelques points, les fonctions ont un fonctionnement identique.

6.9.1 Piles

Les piles peuvent être représentées par des listes LIFO (Last In First Out). Les piles ne sont que des cas particuliers de listes chaînées dont les éléments ne peuvent être ajoutés et supprimés qu'en fin de liste. De ce fait la manipulation s'en trouve grandement simplifiée puisqu'elle ne nécessite que deux fonctions :

- Une fonction pour ajouter un élément au sommet de la pile.
- Une seconde pour le retirer.

6.9.2 Files

Les files sont très similaires aux piles, la seule différence se situe au niveau de l'ajout d'un nouvel élément : il se fait en début de liste. Par contre la suppression se fait toujours en fin de liste. Les files sont aussi appelées listes FIFO (First In First Out). Malgrè un fonctionnement proche de celui des piles, la bibliothèque de gestion des files va être plus complexe, en effet, il faut se déplacer au début ou à la fin de la liste suivant que nous ajoutons ou retirons un élément.

6.10 Classes de mémorisation des variables

Les classes de mémorisation des variables correspondent aux cinq mots-clefs suivants :
typedef
extern
static
auto
register

On a déjà vu **typedef** qui sert à définir un synonyme de type.
typedef n'est en fait associé aux classes de mémorisation que pour des raisons d'ordre syntaxique.

Aux quatre autres classes, à savoir **extern, static, auto** et **register**, sont attachées des propriétés :

- de localisation en machine des variables ;
- de durée de vie des variables ;
- de portée des variables ;
- d'initialisation à la déclaration.

Une classe de mémorisation, lorsqu'elle n'est pas prise par défaut, apparaît avec la déclaration d'une variable sous la forme :
classe de mémorisation type identificateur de variable

6.10.1 Les classes de mémorisation explicites

L'usage explicite de classes de mémorisation a pour vocation de modifier, voire de forcer, les règles de localisation, de durée de vie, de portée et d'initialiser à la déclaration, des variables internes et externes telles qu'elles ont été présentées précédemment dans un contexte implicite par défaut.

extern On peut utiliser la classe de mémorisation “extern” explicite dans deux cas :

1. Dans le même fichier que la variable de définition. Une variable locale ayant la classe de mémorisation “extern” explicite aura en fait les caractéristiques de la variable interne à laquelle elle correspond par son nom : localisation, durée de vie, ...
2. Dans un fichier distinct de la variable de définition. C'est surtout dans ce contexte que l'on utilise la classe de mémorisation “extern” explicite. Cette déclaration “extern” explicite va permettre d'étendre la portée d'une variable externe sur plusieurs fichiers d'un même programme. Si on fait usage de la classe de mémorisation “extern”, deux variables de même nom, déclarées dans deux fichiers distincts donneront lieu à deux définitions distinctes. Aussi ces deux variables seront aussi distinctes l'une de l'autre.

auto Cette classe de mémorisation n'est jamais utilisée explicitement car elle ne peut s'appliquer qu'à des variables locales, lesquelles sont automatiques par défaut.

static La classe de mémorisation “static” peut s'appliquer à une variable externe ou une variable locale.

1. La classe de mémorisation “static” appliquée à une variable externe influence la visibilité de cette variable vis-à-vis d'autres fichiers. Si on souhaite que la variable externe de définition ne puisse être reconnue dans un autre fichier, on lui donne alors la classe explicite “static”. Une fonction a par défaut la qualité externe, on peut donc aussi lui attribuer la classe “static”.
2. La classe de mémorisation “static” appliquée à une variable locale influence directement l'ensemble des caractéristiques de cette variable locale.
   - Localisation : la variable locale sera localisée dans la partie permanente du segment données de la mémoire, comme une variable externe.
   - Durée de vie : elle sera égale au temps du déroulement du programme entier, comme pour une variable externe.
   - Portée : elle ne sera pas modifiée, elle restera limitée à son bloc et aux blocs amboîtés dans ce dernier.
   - Initialisation : elles seront initialisées une seule fois à la compilation. On peut l'initialiser avec une constante ou une expression constante. Si cette initialisation n'apparaît pas explicitement, toute variable arithmétique sera initialisée par défaut avec la valeur zéro et toute variable pointeur avec le pointeur nul.

register Cette classe de mémorisation qui ne peut être appliquée qu'à des variables locales permet, dans une certaine mesure, un accès plus rapide à ces variables et améliore ainsi les performances du programme.

- Localisation : les variables de classe “register” sont prévues pour être localisées dans un registre du processeur.
- Durée de vie : la durée de vie d'une variable de classe “register” est celle d'une variable locale habituelle.
- Initialisation : elle correspond aussi aux variables automatiques.

7 Chaînes de caractères

Il n'y a pas en C à proprement parler de type "chaîne de caractères" ; aussi passe-t-on par les tableaux de caractères ou par les pointeurs sur type caractère pour implémenter les chaînes de caractères.

7.1 Définition

En C, est chaîne de caractères tout tableau (statique ou dynamique) de caractères comportant le caractère '\0'. La valeur d'une chaîne **char s[]** ou **char* s** est la succession des caractères lus de **s[0]==*s** jusqu'à rencontrer le caractère de fin de chaîne '\0'. Pour l'initialisation des chaînes de caractères, se reporter au paragraphe 6.3.2.

Une chaîne, lors de sa saisie sur l'entrée standard ou dans un fichier, ne doit pas, en principe, s'étendre sur plusieurs lignes, sauf précaution particulière : avec beaucoup de compilateurs, cette précaution consiste à faire précéder immédiatement le "retour à la ligne" du caractère "\".

Remarquons que le type pointeur de chaîne de caractères est équivalent au type pointeur de caractère tout court, car un pointeur de chaîne de caractères pointe en fait vers un caractère qui est le premier caractère de la chaîne.

7.2 Fonctions de contrôle et de transformation de caractères

La bibliothèque standard offre, par l'intermédiaire du fichier "en-tête" <**ctype.h**> un certain nombre de fonctions permettant d'une part d'effectuer des contrôles sur des constantes ou des variables de type **char**, d'autre part de transformer une majuscule en minuscule et inversement.

Ces fonctions sont souvent implantées sous forme de macros destinées au préprocesseur, et se trouvent dans le fichier <**ctype.h**>.

Elles comportent un seul argument, variable ou constante, qui est le caractère à tester ou à transformer. La valeur retournée est de type **int**.

7.2.1 Fonctions de contrôle

Chacune de ces fonctions retourne une valeur nulle si le test est négatif, ou une valeur non nulle si le test est positif :

isalnum(c)	teste **c** alphanumérique ?
isalpha(c)	teste **c** alphabétique ?
iscntrl(c)	teste **c** caractère de contrôle ?
isdigit(c)	teste **c** chiffre décimal ?
isgraph(c)	teste **c** caractère visualisable (sauf le caractère "espace") ?
islower(c)	teste **c** lettre minuscule ?
isprint(c)	teste **c** caractère visualisable (caractère "espace" compris) ?
ispunct(c)	teste **c** caractère visualisable qui ne soit ni "espace" ni alphanumérique ?
isspace(c)	teste **c** un des caractères suivants : "espace" \f \n \r \t \v ?
isupper(c)	teste **c** lettre majuscule ?
isxdigit(c)	teste **c** chiffre hexadécimal ?

7.2.2 Fonctions de transformation

tolower(c)	renvoie une lettre **c** en minuscule
toupper(c)	renvoie une lettre **c** en majuscule

7.3 Fonctions de manipulation de chaînes

Nous évoquons ici les principales fonctions de traitement de chaînes de caractères.

7.3.1 Les fonctions de concaténation

- La fonction **strcat(s1,s2)** est une fonction de type pointeur vers une chaîne de caractères. **s1** et **s2** sont deux pointeurs de chaînes de caractères.

  La fonction **strcat** réalise une copie de la chaîne pointée par **s2** à la suite de la chaîne pointée par **s1**.

  Le caractère nul qui, initialement, terminait la chaîne pointée par **s1**, sera écrasé par le premier caractère en provenance de la chaîne pointée par **s2**.

  La valeur retournée par la fonction est l'adresse du premier caractère de la chaîne concaténée, c'est-à-dire la valeur de **s1**.

- La fonction **strncat(s1,s2,n)** est analogue à la fonction **strcat** sauf qu'ici, ce seront au plus les **n** premiers caractères de la seconde chaîne qui seront copiés à la suite de la première chaîne.

7.3.2 Les fonctions de comparaison

- La fonction **strcmp(s1,s2)** est une fonction de type int. Les arguments **s1** et **s2** sont deux pointeurs de chaînes de caractères.

  Si **s1** pointe la chaîne "chaîne1" et **s2** pointe la chaîne "chaîne2", la valeur retournée par la fonction **strcmp** est alors :

positive	si	chaîne1 $>$ chaîne2
nulle	si	chaîne1 $=$ chaîne2
négative	si	chaîne1 $<$ chaîne2

  La valeur numérique exacte retournée n'est pas définie, en dehors du signe, si chaîne1 $\neq$ chaîne2.

  Les critères d'égalité ou d'inégalité de deux chaînes reposent sur les valeurs numériques correspondant aux caractères de chaque chaîne.

  Ces valeurs numériques découlent du code adopté sur son installation (ASCII ou EBCDIC, par exemple).

- La fonction **strncmp(s1,s2,n)** est analogue à la fonction **strcmp**, sauf que ce sont les **n** premiers caractères de chacune des deux chaînes qui seront comparés.

7.3.3 Les fonctions de copie

- La fonction **strcpy(s1,s2)** est une fonction du type pointeur vers une chaîne de caractères. Cette fonction permet de copier une chaîne de caractères pointée par **s2** dans le tableau de caractères pointé par **s1**.

La valeur retournée par la fonction est un pointeur égal à **s1**.

La suite complète des caractères de **s2** est copiée, y compris le caractère nul final.

Remarque : Il est tout à fait nécessaire que le tableau pointé par **s1** ait été défini au préalable, c'est-à-dire qu'il lui corresponde une zone mémoire dûment réservée. Il faut aussi que ce tableau soit assez vaste pour contenir la chaîne qui lui est destinée (caractère nul compris).

- La fonction **strncpy(s1,s2,n)** est analogue à la fonction **strcpy** sauf qu'ici, ce seront **n** caractères au plus de la seconde chaîne qui seront copiés dans le tableau pointé par la première chaîne.

  Si la chaîne pointée par **s2** a une longueur inférieure à **n**, il sera ajouté automatiquement un caractère nul dans la zone réceptrice.

  Le tableau pointé par **s1** devra encore avoir été défini avec une dimension suffisante.

7.3.4 Les fonctions d'indexation

- La fonction **strchr(s,c)** est une fonction du type pointeur vers une chaîne de caractères. L'argument **s** étant du type pointeur de chaîne et **c** un caractère quelconque (ou une valeur entière qui sera convertie en **char**), cette fonction retourne un pointeur pointant vers le premier caractère de la chaîne égal à **c**.

  Si le caractère **c** n'appartient pas à la chaîne pointée par **s**, la fonction **strchr** retourne le pointeur nul.

- La fonction **strrchr(s,c)** est analogue à la fonction **strchr**, sauf que maintenant le pointeur retourné par la fonction pointera vers le dernier caractère **c** rencontré dans la chaîne pointée par **s**.

- La fonction **strstr(s1,s2)** est analogue à la fonction **strchr**, sauf que l'on cherche dans la chaîne pointée par **s1** la première apparition de la chaîne complète pointée par **s2**.

  La valeur retournée par cette fonction, si elle n'est pas nulle, est égale à l'adresse du premier caractère de la chaîne trouvée.

- La fonction **strpbrk(s1,s2)** est analogue à la fonction **strchr** sauf que, maintenant, on cherchera dans la chaîne pointée par **s1** un caractère quelconque de la chaîne pointée par **s2**.

  Remarque : on trouve encore, sur de nombreux compilateurs, la fonction **strchr** sous l'appellation **index**, et la fonction **strrchr** sous l'appellation **rindex**.

7.3.5 Calcul de la longueur d'une chaîne de caractères

La fonction **strlen(s)** retourne une valeur entière égale au nombre de caractères de la chaîne pointée par **s**. Le caractère nul final de la chaîne n'est pas compté.

7.3.6 Les fonctions de conversion

- La fonction **atoi(s)** est du type **int**. L'argument **s** étant un pointeur vers une chaîne de caractères numériques, **atoi(s)** retourne la valeur numérique entière correspondant à cette chaîne.

  Il est nécessaire que la chaîne représente exactement un nombre entier, éventuellement négatif.

- La fonction **atol(s)** est analogue à la fonction **atoi**, sauf que la valeur retournée est de type **long int**.

- La fonction **atof(s)** est analogue à la fonction **atoi**, sauf que la valeur retournée est de type **double**.

  Il est par conséquent nécessaire que la chaîne pointée par **s** représente littéralement une constante flottante.

Pour toutes ces fonctions **atoi**, **atol** et **atof**, si la conversion n'a pu être effectuée correctement, le résultat prend alors une valeur quelconque, indéterminée.

8 La bibliothèque mathématique du C

On fait appel à cette bibliothèque par la directive `#include <math.h>`. Les fonctions de cette bibliothèque sont de type **double**. Il en est de même des paramètres.

Classe de fonction	*Expression en C*	*Expression mathématique*
Trigonométrique	**acos(x)**	Arc cos x
	asin(x)	Arc sin x
	atan(x)	Arc tg x
	cos(x)	cos x
	sin(x)	sin x
	tan(x)	tg x
Hyperbolique	**cosh(x)**	ch x
	sinh(x)	sh x
	tanh(x)	th x
Exponentielle	**exp(x)**	e^x
Logarithmique	**log(x)**	ln x
	log10(x)	log_{10} x
Reste de la division	**fmod(x,y)**	résidu de x modulo y
Puissance	**pow(x,y)**	x^y
Racine carrée	**sqrt(x)**	$\sqrt{x}$
Valeur absolue	**fabs(x)**	$\|x\|$
Approche entière	**ceil(x)**	Approche par excès
	floor(x)	Approche par défaut

Remarques :

- La fonction `rand()` du type **int** retourne un nombre compris entre 0 et **RAND_MAX**. **RAND_MAX** est le plus grand nombre entier que l'on pourrait obtenir avec la fonction `rand()`. Plusieurs appels successifs à la fonction `rand()` permettraient d'obtenir une suite de nombres entiers pseudo-aléatoires. Cett fonction s'utilise sans paramètre.

- En utilisant la bibliothèque `<time.h>`, nous avons à notre disposition la fonction `time()` qui retourne la valeur de la date et de l'heure actuelle (l'origine de temps est généralement le 1^{er} janvier 1970 à 0 heures G.M.T. La fonction `difftime(tfinal,tinitial` retourne la différence entre les deux temps. Enfin la fonction `ctime()` permet de convertir une valeur entière numérique, susceptible de représenter la date et l'heure en une chaîne de caractères plus explicites.

9 Organisation d'un programme en fichiers : programmation modulaire

9.1 Portée et durée de vie des variables

9.1.1 Portée

Une variable définie à l'intérieur d'un bloc (notamment, à l'intérieur d'une fonction) est locale à ce bloc : avant d'y entrer et après en être sorti, cette variable n'existe pas. Ces variables sont dites *locales*.
Une variable définie en dehors d'un bloc, au niveau du fichier même, est connue de tout le fichier (du moins depuis sa définition). Selon sa déclaration, cette variable peut également être connue des autres fichiers : si elle est déclarée `int taille` ou `extern int taille` (on considère par défaut qu'une variable est déclarée comme **extern**), alors elle devient potentiellement globale puisque tout fichier peut s'y référer, du moment qu'il déclare lui-même "`extern int taille;`". Pour restreindre la portée de la variable au fichier même, il faut la déclarer *statique* : "`static int taille;`".

9.1.2 Durée de vie

Généralement, la durée de vie d'une variable est son bloc de définition (son fichier si la variable est globale au fichier) : c'est ce qu'on appelle une *variable de classe d'allocation automatique*. Cependant, on peut rallonger la durée de vie d'une variable locale (les variables globales étant connues depuis leur définition jusqu'à la fin de l'exécution du programme) en la déclarant **static**. En effet, une variable i locale à une fonction donnée existera (bien que ne restant accessible que de sa fonction de définition) du premier appel de sa fonction de définiton à la fin de l'exécution du programme. Ainsi, d'un appel à l'autre, on peut se référer à la dernière valeur de la variable, car c'est une variable locale dont l'emplacement mémoire est réservé une fois pour toute, de la première apparition de la variable jusqu'à la fin du programme.

9.2 Entêtes, sources et bibliothèques

Lorque l'on programme, on identifie différents composants, par type de donnée manipulée (tableau, file, arbre...), ou encore par type de traitement fonctionnel ou technique (gestion des accès en base de données, fonctions de calcul évoluées, multi-threading etc.).

Ce premier découpage permet une meilleure organisation (en forï¿½ant l'organisation), une réutilisation des composants (différents programmes peuvent nécessiter la modélisation d'une file ; un composant d'accès à une base de données est lié à une architecture et non à un programme particulier), enfin une compilation efficace (je n'ai pas besoin de recompiler les modules "file" et "fonction de calcul" si seul le fichier "accès base de données" a changé).

Un module lui-même est divisé en deux parties : son mode d'emploi (le fichier à entête d'extension ".h" qui déclare les variables et les fonctions prototypées) et sa définition même (le fichier source d'extension ".c" qui code les fonctions associées).

En effet, le seul moyen dont dispose l'auteur d'un module A pour s'assurer que les autres modules qui forment un programme utilisent correctement les variables et fonctions qu'il rend publiques consiste à écrire un fichier en-tête (fichier A.h) contenant toutes les déclarations publiques. Ce fichier doit être inclus par la directive `#include` dans le module qui implante les objets publics (fichier A.c) et dans chacun des modules qui les utilisent. De cette manière tous ces fichiers "voient" les mêmes définitions de types, les

mêmes déclarations de variables et les mêmes prototypes de fonctions ; ces déclarations sont écrites en un seul endroit, et toute modification de l'une d'entre elles se répercute sur tous les fichiers qui en dépendent.

La nécessitï¿½ de ces fichiers en-tête apparaît encore plus grande quand on considère le cas des bibliothèques, c'est-à-dire des modules que leurs fonctionnalités placent en position de prestataires de services vis-à-vis des autres modules qui composent un programme ; on parle alors de module serveur et de modules clients. En fait, on peut presque toujours voir la modularité en termes de serveurs et clients, car il y a toujours une hiérarchie parmi les modules. Le propre des bibliothèques est d'être conçues de manière indï¿½pendante des clients, afin de pouvoir être utilisées dans un programme présent et un nombre quelconque de programmes futurs. L'intérêt de leur associer le meilleur dispositif pour minimiser le risque de mauvaise utilisation est évident.

Typiquement, un fichier en-tête comportera les éléments suivants :

- Des directives `#include` concernant les autres fichiers en-tête nécessaires pour la compréhension (par le compilateur) des éléments qui apparaissent dans le fichier en-tête en question.
- Des définitions de constantes, soit sous forme de directives `#define` soit sous forme de type énuméré, qui sont des informations symboliques échangées entre le serveur et ses clients. Exemple : dans une bibliothèque graphique, les noms conventionnels des couleurs.
- Des définitions de structures (struct, union) et de types (typedef) qui définissent la nature des objets manipulés par la bibliothèque. Typiquement, ces types permettent aux clients de déclarer les objets qui sont les arguments et les résultats des fonctions de la bibliothèque. Exemple : dans une bibliothèque graphique, la définition de types point, segment, etc.
- Les déclarations "extern" des variables publiques du serveur. Les définitions correspondantes (sans le qualifieur extern) figureront dans le module serveur. Remarque : l'emploi de variables publiques est déconseillé ; un module ne devrait offrir que des fonctions.
- Les déclarations des fonctions publiques du serveur. Les définitions correspondantes figureront dans le module serveur. En syntaxe originale seules les déclarations des fonctions qui ne rendent pas un entier sont nécessaires, mais même dans ce cas c'est une bonne habitude que d'y mettre les déclarations de toutes les fonctions, cela constitue un germe de documentation.

Bien entendu, tous les noms de variables et fonctions du module serveur qui ne figurent pas dans le fichier en-tête doivent être rendus privés (en les qualifiant static).

S'il est utile qu'un client puisse consulter ou modifier une variable du serveur, écrivez une fonction qui ne fait que cela (mais en contrôlant la validité de la consultation ou de la modification). C'est bien plus sûr que de donner libre accès à la variable.

Enfin, on peut utitiliser des bibliothèques plus complexes que les simples fichiers à entête (qu'ils soient standards ou personnalisés) : ce sont les *bibliothèques compilées*. Elles correspondent à des modules qui sont groupés et précompilés : on n'a dès lors plus accès aux fichiers source, ni à entête, ni même aux objets, mais à un conglomérat **.a** qui joue à la fois le rôle des entêtes **.h** et des objets **.o**.
C'est sous la forme de librairies (traduction littérale et erronée du terme anglais "library", mais néanmoins usitée) que sont proposés de nombreux utiliaires du commerce (**dll** sous Windows).

9.3 Fonction "main"

La signature de la fonction **main** n'est pas libre. Tout d'abord, **main** doit retourner un type **void** ou **int** (une mise en garde apparaît à la compilation lorsque l'on renvoie **void**). Il est toujours conseillé de retourner quelque chose, ne serait-ce que le statut de réussite ou d'échec du déroulement du programme.

Ensuite, les arguments : si le programme ne prend pas d'argument, la fonction **main** n'en prend pas non plus et l'on déclare `void main()` ou `int main(void)` ; en revanche, si le programme prend des arguments, alors il faut déclarer `void main(int argc, char* argv[])` ou `int main(int argc, char* argv[])`. L'argument **argc** est un entier qui donne la longueur du tableau **char* argv[]** et cette longueur est le nombre des paramètres effectifs du programme *plus 1* ; l'argument **char* argv[]** contient quant à lui la liste des arguments du programme, *plus* le nom de la commande. Ici, il faut dissocier les arguments de la fonction C et les arguments du programme lors de son appel depuis une console : l'appel sera `nomBinaire` $\texttt{arg}_1$ $\texttt{arg}_2$ ... $\texttt{arg}_n$ tandis que les arguments de la fonction **main** seront toujours **argc** et **argv**, où **argc** vaut $n+1$ et où le tableau **argv** contient exactement la ligne de commande `nomBinaire` $\texttt{arg}_1$ $\texttt{arg}_2$ ... $\texttt{arg}_n$, c'est-à-dire : `argv[0] = nomBinaire, argv[1] =` $\texttt{arg}_1$`, argv[2] =` $\texttt{arg}_2$ ... `argv[n] =` $\texttt{arg}_n$.

```
//programme qui fait la multiplication d'une liste d'entiers
int main(int argc, char *argv[])
{
    int i;
    int res = 1;
    for (i = 1; i <= argc; i++)
       res = res * argv[i];
    printf("Resultat = %d\n", res);
    return 0;
}
```

Attention ! ! ! - Puisque les arguments ne sont pas décrits explicitement, si l'on s'attend à des paramètres particuliers (ex. : un nom de fichier d'entrée et un nom de fichier de sortie), il faut dans la fonction **main** inclure une procédure de vérification des arguments qui sont passés à l'appel.

```
//programme qui extrait des données d'un fichier pour les placer dans un fichier de sortie
int main(int argc, char *argv[])
{
    FILE* f_entree = NULL;
    FILE* f_sortie = NULL;
    if (argc != 3)
    {
       fprintf(sdterr, "Nombre d'arguments incorrects\n");
       return 1; //on renvoie une valeur non nulle pour signifier l'erreur
    }
    f_entree = fopen(argv[1], "r");
    if (!  f_entree)
    {
       fprintf(sdterr, "Impossible d'ouvrir le fichier %s en lecture\n", argv[1]);
       return 1; //on renvoie une valeur non nulle pour signifier l'erreur
    }
    f_sortie = fopen(argv[2], "w");
    if (!  f_sortie)
    {
```

```
        fprintf(sdterr, "Impossible d'ouvrir le fichier %s en écriture\n", argv[2]);
        return 1; //on renvoie une valeur non nulle pour signifier l'erreur
    }
    ...  //traitement
    if (f_entree)
    {
        fclose(f_entree);
        f_entree = NULL;
    }
    if (f_sortie)
    {
        fclose(f_sortie);
        f_sortie = NULL;
    }
    return 0; //sortie sans erreur
}
```

9.4 Commande de compilation

9.4.1 En ligne

Compiler un fichier unique : "`gcc usager.c`"
Attention ! ! ! - Ce fichier doit contenir une fonction `main`, point d'entrée de tout programme C.
Cette commande a pour effet de compiler le fichier "usager.c" ; de cette compilation, si elle réussit, résulte un fichier exécutable (*binaire*), du nom de "a.out". Pour décider du nom du binaire, il faut utiliser l'option "**-o**" : "`gcc -o usager.c usager`". Un fichier exécutable du nom de "usager" est alors créé, hors erreur de compilation. D'autres options utiles sur **gcc** : "**gcc -Wall**" pour afficher *tous* les messages d'avertissement, "**gcc -ansi**" pour compiler en C ANSI.

Un programme peut maintenant être constitué de plusieurs fichiers. Supposons par exemple que notre fichier "usager.c" manipule des listes chaînées de type *pile* dont le comportement est décrit dans un fichier "pile.c", alors "usager.c" a besoin des fonctions de "pile.c" pour compiler. Dans ce genre de configuration, il faut :
1) - pour compiler "usager.c", y inclure la bibliothèque "pile.h" que l'on n'aura pas manqué de créer ;
2) - pour générer l'exécutable final "usager", avoir au préalable généré les objets "pile.o" et "usager.o" (la compilation préalable de "usager.o" n'est pas obligatoire, mais il est toujours plus efficace de découper au maximum la compilation).

Pour générer un objet à partir d'un fichier source, il faut utiliser l'option "**-c**" : "`gcc -c pile.c`". Un composant du nom de "pile.o" est alors créé. Notons que l'on peut encore spécifier le nom du fichier output par l'option "**-o**", mais il est préférable de nommer l'objet du même nom que le fichier source, avec l'extension "**.o**" ; il est d'ailleurs même recommandé de spécifier malgré tout explicitement ce nom par défaut :

```
>gcc -c pile.c -o pile.o
>gcc -c usager.c -o usager.o
```

Une fois les objets créés, on peut passer à la phase de génération du programme global, qui se réduit

à l'édition de liens. En effet, pour l'instant, d'une part "usager.o" sait qu'il existe des fonctions de manipulation de pile définies quelque part (grâce à l'inclusion de "pile.h" dans "usager.c"), d'autre part ces fonctions ont été compilées dans "pile.o" ; il reste donc au compilateur à lier les différents modules, les appels et les définitons de fonctions à partir de la fonction **main** ; c'est ce qu'il fait lorsqu'on lui indique : "`gcc -o usager pile.o usager.o`"
Le compilateur **gcc** génére alors un exécutable "usager" à partir des objets "pile.o" et "usager.o". Ici il ne faut pas spécifier l'option "**-c**", car cette option signifie précisément de compiler sans chercher à faire l'édition de liens... or, c'est justement ce que l'on cherche à faire ici !

Remarque : Pour compiler le programme dans son intrégralité, on peut regrouper ces commandes à l'intérieur d'un fichier texte (ex., "usager.gcc"), rendre ce fichier exécutable, puis simplement l'exécuter par la ligne de commande : "usager.gcc".

Pour faire appel à des éléments d'une librairie, au moment de l'édition de liens :
`gcc -o usager pile.o usager.o -l nomLibrairie.a`

L'option **-l** recherche la librairie `nomLibrairie.a` dans le répertoire standard, tandis que l'option **-L** la recherche dans les répertoires personnalisés qui sont listés dans la variable d'environnement `LD_LIBRARY_PATH` (la variable `LD_LIBRARY_PATH` joue pour les bibiothèques le rôle que joue la variable `PATH` pour les commandes).

9.4.2 Utilisation de la commande make

Les Makefiles sont des fichiers, généralement appelés makefile ou Makefile, utilisés par le programme "make" pour exécuter un ensemble d'actions, comme la compilation d'un projet, l'archivage de document, la mise à jour de site, ... Cette section présentera le fonctionnement de makefile au travers de la compilation d'un petit projet en C.

Attention, il existe une multitude d'utilitaires de Makefile fonctionnant sur différents systèmes (gmake, nmake, tmake, ...). Les Makefiles n'étant malheureusement pas normalisés, certaines syntaxes ou fonctionnalités peuvent ne pas fonctionner sur certains utilitaires. Nous nous basons ici sur l'utilitaire GNU make. Toutefois les notions abordées devraient être utilisables avec la majorité des utilitaires.

Le projet exemple : "hello world" Le projet utilisé ici sera le classique "hello world" découpé en trois fichiers :

- hello.c

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

void Hello(void)
{
  printf("Hello World\n");
}
```

- hello.h

```
#ifndef H_GL_HELLO
```

```
#define H_GL_HELLO

void Hello(void);

#endif
```

- main.c

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

int main(void)
{
  Hello();
  return EXIT_SUCCESS;
}
```

Présentation de Makefile Un Makefile est un fichier constitué de plusieurs règles de la forme :

```
cible: dependance
  commandes
```

Chaque commande est précédée d'une tabulation. Lors de l'utilisation d'un tel fichier via la commande make la première règle rencontrée, ou la règle dont le nom est spécifié, est évaluée. L'évaluation d'une règle se fait en plusieurs ï¿½tapes :

- Les dépendances sont analysées, si une dépendance est la cible d'une autre règle du Makefile, cette règle est à son tour évaluée.
- Lorsque l'ensemble des dépendances est analysé et si la cible ne correspond pas à un fichier existant ou si un fichier dépendance est plus récent que la règle, les différentes commandes sont exécutées.

1. Makefile minimal :

```
hello: hello.o main.o
  gcc -o hello hello.o main.o

hello.o: hello.c
  gcc -o hello.o -c hello.c -W -Wall -ansi -pedantic

main.o: main.c hello.h
  gcc -o main.o -c main.c -W -Wall -ansi -pedantic
```

Regardons de plus près sur cet exemple comment fonctionne un Makefile : Nous cherchons à créer le fichier exécutable hello, la première dépendance est la cible d'une des règles de notre Makefile,

nous évaluons donc cette règle. Comme aucune dépendance de hello.o n'est une règle, aucune autre règle n'est à évaluer pour compléter celle-ci. Deux cas se présentent ici : soit le fichier hello.c est plus récent que le fichier hello.o, la commande est alors exécutée et hello.o est construit, soit hello.o est plus récent que hello.c et la commande n'est pas exécutée. L'évalution de la règle hello.o est terminée. Les autres dépendances de hello sont examinées de la même manière puis, si nécessaire, la commande de la règle hello est exécutée et hello est construit.

2. Makefile enrichi : Plusieurs cas ne sont pas gérés dans l'exemple précédent :

   - Un tel Makefile ne permet pas de générer plusieurs exécutables distincts.
   - Les fichiers intermédiaires restent sur le disque dur même lors de la mise en production.
   - Il n'est pas possible de forcer la regénération intégrale du projet

   Ces différents cas conduisent à l'écriture de règles complémentaires :

   - all : généralement la première règle du fichier, elle regroupe dans ces dépendances l'ensemble des exécutables à produire.
   - clean : elle permet de supprimer tous les fichiers intermédiaires.
   - mrproper : elle supprime tout ce qui peut être regénéré et permet une reconstruction complète du projet.

   En ajoutant ces règles complémentaires, notre Makefile devient donc :

```
all: hello

hello: hello.o main.o
  gcc -o hello hello.o main.o

hello.o: hello.c
  gcc -o hello.o -c hello.c -W -Wall -ansi -pedantic

main.o: main.c hello.h
  gcc -o main.o -c main.c -W -Wall -ansi -pedantic

clean:
  rm -rf *.o

mrproper: clean
  rm -rf hello
```

Définition de variables

1. Variables personnalisées : Il est possible de définir des variables dans un Makefile, ce qui rend les évolutions bien plus simples et plus rapides, en effet plus besoin de changer l'ensemble des règles si le compilateur change, seule la variable correspondante est à modifier. Une variable se déclare sous

la forme NOM=VALEUR et se voit utiliser via $(NOM). Nous allons donc définir quatre variables dans notre Makefile :

- Une désignant le compilateur utilisé nommée CC (une telle variable est typiquement nommée CC pour un compilateur C, CPP pour un compilateur C++).
- CFLAGS regroupant les options de compilation (généralement cette variable est nommée CFLAGS pour une compilation en C, CXXFLAGS pour le C++).
- LDFLAGS regroupant les options de l'édition de liens.
- EXEC contenant le nom des exécutables à générer.

Nous obtenons ainsi :

```
CC=gcc
CFLAGS=-W -Wall -ansi -pedantic
LDFLAGS=
EXEC=hello

all: $(EXEC)

hello: hello.o main.o
  $(CC) -o hello hello.o main.o $(LDFLAGS)

hello.o: hello.c
  $(CC) -o hello.o -c hello.c $(CFLAGS)

main.o: main.c hello.h
  $(CC) -o main.o -c main.c $(CFLAGS)

clean:
  rm -rf *.o

mrproper: clean
  rm -rf $(EXEC)
```

2. Variables internes : Il existe plusieurs variables internes au Makefile, citons entre autres :

- $@ : Le nom de la cible
- $< : Le nom de la première dépendance
- $^ : La liste des dépendances
- $? : La liste des dépendances plus récentes que la cible
- $* : Le nom du fichier sans suffixe

Notre Makefile ressemble donc maintenant à :

```
CC=gcc
CFLAGS=-W -Wall -ansi -pedantic
LDFLAGS=
EXEC=hello

all: $(EXEC)

hello: hello.o main.o
  $(CC) -o $@ $^ $(LDFLAGS)

hello.o: hello.c
  $(CC) -o $@ -c $< $(CFLAGS)

main.o: main.c hello.h
  $(CC) -o $@ -c $< $(CFLAGS)

clean:
  rm -rf *.o

mrproper: clean
  rm -rf $(EXEC)
```

Les règles d'inférence Makefile permet également de créer des règles génériques (par exemple construire un .o à partir d'un .c) qui se verront appelées par défaut. Une telle règle se présente sous la forme suivante :

```
%.o: %.c
  commandes
```

Il existe une autre notation plus ancienne de cette règle :

```
.c.o:
  commandes
```

Il devient alors aisé de définir des règles par défaut pour générer nos différents fichiers :

```
CC=gcc
CFLAGS=-W -Wall -ansi -pedantic
LDFLAGS=
EXEC=hello

all: $(EXEC)

hello: hello.o main.o
  $(CC) -o $@ $^ $(LDFLAGS)
```

```
%.o: %.c
  $(CC) -o $@ -c $< $(CFLAGS)

clean:
  rm -rf *.o

mrproper: clean
  rm -rf $(EXEC)
```

Comme le montre clairement l'exemple précédent, main.o n'est plus reconstruit si hello.h est modifié. Il est possible de préciser les dépendances séparément des règles d'inférence et de rétablir le fonctionnement original, pour obtenir finalement :

```
CC=gcc
CFLAGS=-W -Wall -ansi -pedantic
LDFLAGS=
EXEC=hello

all: $(EXEC)

hello: hello.o main.o
  $(CC) -o $@ $^ $(LDFLAGS)

main.o: hello.h

%.o: %.c
  $(CC) -o $@ -c $< $(CFLAGS)

clean:
  rm -rf *.o

mrproper: clean
  rm -rf $(EXEC)
```

La cible .PHONY Dans l'exemple, clean est la cible d'une règle ne présentant aucune dépendance. Supposons que clean soit également le nom d'un fichier présent dans le répertoire courant, il serait alors forcément plus récent que ses dépendances et la règle ne serait alors jamais exécutée. Pour pallier ce problème, il existe une cible particulière nommée .PHONY dont les dépendances seront systématiquement reconstruites. Nous obtenons donc :

```
CC=gcc
CFLAGS=-W -Wall -ansi -pedantic
LDFLAGS=
EXEC=hello
```

```
all: $(EXEC)

hello: hello.o main.o
  $(CC) -o $@ $^ $(LDFLAGS)

main.o: hello.h

%.o: %.c
  $(CC) -o $@ -c $< $(CFLAGS)

.PHONY: clean mrproper

clean:
  rm -rf *.o

mrproper: clean
  rm -rf $(EXEC)
```

Génération de la liste des fichiers objets Plutôt que d'énumérer la liste des fichiers objets dans les dépendances de la règle de construction de notre exécutable, il est possible de la générer automatiquement à partir de la liste des fichiers sources. Pour cela nous rajoutons deux variables au Makefile :

- SRC qui contient la liste des fichiers sources du projet.
- OBJ pour la liste des fichiers objets.

La variable OBJ est remplie à partir de SRC de la manière suivante :

```
OBJ= $(SRC:.c=.o)
```

Pour chaque fichier .c contenu dans SRC nous ajoutons à OBJ un fichier de même nom mais portant l'extension .o. Nous obtenons alors :

```
CC=gcc
CFLAGS=-W -Wall -ansi -pedantic
LDFLAGS=
EXEC=hello
SRC= hello.c main.c
OBJ= $(SRC:.c=.o)

all: $(EXEC)

hello: $(OBJ)
  $(CC) -o $@ $^ $(LDFLAGS)
```

```
main.o: hello.h

%.o: %.c
  $(CC) -o $@ -c $< $(CFLAGS)

.PHONY: clean mrproper

clean:
  rm -rf *.o

mrproper: clean
  rm -rf $(EXEC)
```

Construction de la liste des fichiers sources De la même manière, il peut être utile de gérer la liste des fichiers sources de manière automatique (attention toutefois à la gestion des dépendances vis à vis des fichiers d'entête). Pour ce faire nous allons faire remplir la variable SRC avec les différents fichiers .c du répertoire. La premiï¿½re idée qui vient généralement pour réaliser cette tâche est l'utilisation du joker *.c, malheureusement il n'est pas possible d'utiliser ce joker directement dans la définition d'une variable. Nous devons utiliser la commande wildcard qui permet l'utilisation des caractères joker. Le Makefile correspondant est :

```
CC=gcc
CFLAGS=-W -Wall -ansi -pedantic
LDFLAGS=
EXEC=hello
SRC= $(wildcard *.c)
OBJ= $(SRC:.c=.o)

all: $(EXEC)

hello: $(OBJ)
  $(CC) -o $@ $^ $(LDFLAGS)

main.o: hello.h

%.o: %.c
  $(CC) -o $@ -c $< $(CFLAGS)

.PHONY: clean mrproper

clean:
  rm -rf *.o

mrproper: clean
  rm -rf $(EXEC)
```

Commandes silencieuses Lorsque le nombre de règles d'un Makefile augmente, il devient très rapidement fastidieux de trouver les messages d'erreur affichés au milieu des lignes de commandes. Les Makefiles permettent de désactiver l'écho des lignes de commandes en rajoutant le caractï¿½re @ devant la ligne de commande, notre Makefile devient alors :

```
CC=gcc
CFLAGS=-W -Wall -ansi -pedantic
LDFLAGS=
EXEC=hello
SRC= $(wildcard *.c)
OBJ= $(SRC:.c=.o)

all: $(EXEC)

hello: $(OBJ)
  @$(CC) -o $@ $^ $(LDFLAGS)

main.o: hello.h

%.o: %.c
  @$(CC) -o $@ -c $< $(CFLAGS)

.PHONY: clean mrproper

clean:
  @rm -rf *.o

mrproper: clean
  @rm -rf $(EXEC)
```

Les Makefiles conditionnels Les Makefiles nous offrent en plus une certaine souplesse en introduisant des directives, assez proches des directives de compilation du C, qui permettent d'exécuter conditionnellement une partie du Makefile en fonction de l'existence d'une variable, de sa valeur, ... Supposons, par exemple, que nous souhaitions compiler notre projet tantôt en mode debug, tantôt en mode release sans avoir à modifier plusieurs lignes du Makefile pour passer d'un mode à l'autre. Il suffit de créer une variable DEBUG et tester sa valeur pour changer de mode :

```
DEBUG=yes
CC=gcc
ifeq ($(DEBUG),yes)
  CFLAGS=-W -Wall -ansi -pedantic -g
  LDFLAGS=
else
  CFLAGS=-W -Wall -ansi -pedantic
  LDFLAGS=
```

```
endif
EXEC=hello
SRC= $(wildcard *.c)
OBJ= $(SRC:.c=.o)

all: $(EXEC)
ifeq ($(DEBUG),yes)
  @echo "Gï¿½nï¿½ration en mode debug"
else
  @echo "Gï¿½nï¿½ration en mode release"
endif

hello: $(OBJ)
  @$(CC) -o $@ $^ $(LDFLAGS)

main.o: hello.h

%.o: %.c
  @$(CC) -o $@ -c $< $(CFLAGS)

.PHONY: clean mrproper

clean:
  @rm -rf *.o

mrproper: clean
  @rm -rf $(EXEC)
```

Dans ce cas l'exécutable est généré en mode debug, il suffit de changer la valeur de la variable DEBUG pour revenir en mode release.

sous-Makefiles Il arrive parfois que l'on souhaite créer plusieurs Makefiles correspondant à des parties distinctes d'un projet (par exemple : client/serveur, bibliothèques de fonctions, sources réparties dans plusieurs répertoires, ...). Toutefois il est souhaitable que ces Makefiles soient appelés par un unique Makefile "maître" et que les options soient identiques d'un Makefile à l'autre. Il est ainsi possible d'appeler un Makefile depuis un autre Makefile grâce à la variable $(MAKE) et de fournir à ce second Makefile des variables définies dans le premier en exportant celles-ci via l'instruction export, avant d'invoquer le second Makefile. Voyons cela à travers notre petit projet, nous supposons qu'il se situe dans le sous-répertoire hello et que le Makefile maître, situé dans le répertoire principal, définira le compilateur et les options utilisées ; nous obtenons alors le Makefile maître suivant :

```
export CC=gcc
export CFLAGS=-W -Wall -ansi -pedantic
export LDFLAGS=
HELLO_DIR=hello
EXEC=$(HELLO_DIR)/hello
```

```
all: $(EXEC)

$(EXEC):
  @(cd $(HELLO_DIR) && $(MAKE))

.PHONY: clean mrproper $(EXEC)

clean:
  @(cd $(HELLO_DIR) && $(MAKE) $@)

mrproper: clean
  @(cd $(HELLO_DIR) && $(MAKE) $@)
```

et la Makefile suivant :

```
EXEC=hello
SRC= $(wildcard *.c)
OBJ= $(SRC:.c=.o)

all: $(EXEC)

hello: $(OBJ)
  @$(CC) -o $@ $^ $(LDFLAGS)

%.o: %.c
  @$(CC) -o $@ -c $< $(CFLAGS)

.PHONY: clean mrproper

clean:
  @rm -rf *.o

mrproper: clean
  @rm -rf $(EXEC)
```

9.5 Ce qu'il faut retenir

Le découpage des sources en modules présente différents avantages :
1 - en forçant l'organisation, il force aussi l'analyse fonctionnelle (isoler des chaînes de traitements) et technique (architecture) du projet ;
2 - en rendant indépendante les différentes parties du code, il en rend la compilation bien plus efficace ;
3 - en identifiant des modules par fonction (technique, de communication, de structure abstraite de donnée), il rend le code réutilisable.

10 Gestion des Entrées/Sorties

10.1 Les fonctions d'Entrées/Sorties non formatées

10.1.1 Lecture

- Lecture de caractère : **getchar()**
  `char c; c = getchar();` permet de lire un caractère sur l'entrée standard et de le récupérer dans une variable de type caractère.

- Lecture d'une chaîne de caractères : **gets(s)**
  s est le nom d'un tableau (ou d'un pointeur vers un tableau), tableau déclaré antérieurement, et qui recevra la chaîne de caractères lue.
  De façon plus générale, **s** peut être le nom d'un pointeur vers le tableau récepteur.

  Un caractère nul est ajouté automatiquement à la suite de la chaîne quand on termine l'entrée de celle-ci par un "retour à la ligne".

  Remarque : **gets** est une fonction de type pointeur de caractère qui retourne le pointeur **s** si l'opération de lecture s'est déroulée normalement, et le pointeur nul dans le cas contraire.

Remarque : la constante EOF, définie dans une librairie standard, signifie "End Of File".

10.1.2 Ecriture

- Ecriture de caractère : **putchar()**
  `char c = 'a'; putchar(c);` permet d'écrire sur la sortie standard une variable de type caractère.

- Ecriture d'une chaîne de caractères : **puts(s)**
  s est le nom d'un tableau qui contient la chaîne de caractères.
  De façon plus générale, **s** peut être un pointeur vers une chaîne de caractères, ou la chaîne elle-même.

  Remarque : La fonction **puts** provoque automatiquement un retour à la ligne après l'édition de la chaîne de caractères.

  C'est donc le caractère '\0' final qui est transformé automatiquement en '\n'

  La fonction **puts** est de type **int**. Elle retourne la valeur zéro si l'opération d'écriture s'est déroulée normalement, et une valeur non nulle dans le cas contraire.

10.2 Les fonctions d'Entrées/Sorties formatées

Les données lues, ou à imprimer, seront converties selon le format particulier choisi.

10.2.1 La fonction de lecture

scanf(`"chaîne de contrôle", argument1, argument2, ...);`
La chaîne de contrôle indique les formats relatifs aux arguments, selon l'ordre de ces arguments dans la liste. Chaque argument est un pointeur vers l'objet qui recevra la donnée lue, après conversion de celle-ci.
Dans la chaîne de contrôle, le caractère "%" introduit une spécification de format.
Indiquons les principales spécifications de **scanf**.
Dans le tableau suivant sont représentés :

- les spécifications de formats acceptés par la fonction **scanf** ;

- la forme de donnée constante attachée à chacun de ces formats ;
- le type de l'objet, pointé par l'argument, correspondant à une spécification de format.

Format	*Données constantes*	*Types d'objets pointés*
d	décimale entière signée	**int**
hd	décimale entière signée	**short int**
ld	décimale entière signée	**long int**
u	décimale entière non signée	**int**
hu	décimale entière non signée	**short int**
lu	décimale entière non signée	**long int**
o	octale	**int**
ho	octale	**short int**
lo	octale	**long int**
x	hexadécimale	**int**
hx	hexadécimale	**short int**
lx	hexadécimale	**long int**
f	flottante ou entière	**float**
lf	flottante ou entière	**double**
c	caractère	**char**
s	chaîne de caractères	tableau de type **char**

Remarques :

- Si le format est précédé du caractère "*", cela signifie qu'il n'est pas prévu d'argument pour la donnée particulière qui est impliquée. Aussi, cette donnée à lire sera-t-elle sautée.
- Les données à entrer sont naturellement séparées par des blancs.
  Toutefois, on peut fixer l'étendue du champ, en nombre de caractères, de chaque donnée à lire. Pour cela, on indique avant le format l'étendue de ce champ, à l'aide d'une constante entière.
  Exemple : "%3s" : on lira une chaîne de 3 caractères.
  "%4d" : le nombre entier à lire s'étend sur 4 caractères (signe inclus, s'il existe).
- Il est nécessaire de respecter scrupuleusement le type de l'objet attaché à un format car la fonction **scanf** n'assure pas de contrôle.
- La fonction **scanf** est de type **int**. Normalement, cette fonction retourne le nombre d'articles effectivement lus. En cas de fin de fichier, elle retourne la valeur de la macro du préprocesseur désignée par **EOF**. En cas de problème au sujet de la conversion, elle retourne la valeur zéro.
- Avec la spécification de format **s**, le compilateur ajoute automatiquement un caractère '\0' à la suite de la chaîne lue. Ce caractère sera placé dans le tableau récepteur.

10.2.2 La fonction d'impression

printf(`"chaîne de contrôle", argument1, argument2, ...);` La chaîne de contrôle contient le texte à afficher (caractères de contrôle compris) et les spécifications de formats correspondantes respectivement à chaque argument de la liste.

Les arguments de la liste sont des constantes ou des identificateurs d'objets ou des expressions.

On peut ne pas mettre de liste d'arguments. Dans ce cas, il n'y aura aucune spécification de format dans la chaîne de contrôle.

Les principales spécifications de **printf** sont :

Format	*Données imprimées*	*Types d'objets*
d	décimale entière signée	integral
u	décimale entière non signée	integral
o	octale	integral
x	hexadécimale entière non signée lettres en minuscules	integral
X	hexadécimale entière non signée lettres en majuscules	integral
f	décimale flottante forme virgule fixe	**float** ou **double**
e ou **E**	décimale flottante forme virgule flottante	**float** ou **double**
g ou **G**	forme la plus réduite de **f** ou **e**	**float** ou **double**
c	caractère	**char**
s	chaîne de caractères	tableau ou pointeur

Remarques :

- On peut préciser la largeur minimale de chaque champ en faisant précéder le format d'une constante entière :
  "%10d" : 10 caractères ou moins seront réservés pour imprimer la donnée entière concernée. La donnée est cadrée à droite du champ.
- Un signe "-" avant le format indique que la donnée sera cadrée à gauche du champ.
- Pour les données de type chaîne de caractères, le "." avant la constante indique le nombre maximum de caractères à imprimer :
  "%-30.8s" : il sera réservé en tout 30 caractères pour la chaîne mais les 8 premiers caractères seulement seront imprimés et cadrés à gauche.
- Pour les données de type **float** ou **double**, le point indique le nombre de chiffres décimaux aprés la virgule qu'il faudra imprimer :
  "%.12f" : il sera imprimé 12 chiffres après la virgule.

10.3 Les fichiers

10.3.1 E/S standards

Entrée standard == **stdin** (par défaut, le clavier)
Sortie standard == **stdout** (par défaut, l'écran)
Erreur standard == **stderr** (par défaut, l'écran)

Mais on peut vouloir lire / écrire ailleurs, dans un fichier spécifique : lire un fichier de paramètres, écrire un journal des erreurs etc...

10.3.2 Le type FILE

Le type FILE, contrairement à sa dénomination, ne désigne pas exactement un fichier, mais un *flot des données qui transitent par la mémoire tampon* (buffer). Les flux **stdin**, **stdout** et **stderr** sont tous les 3 des pointeurs sur **FILE**. Pour manipuler un fichier, on déclarera également une variable de type **FILE***, qui permettra d'avoir un curseur sur le fichier que l'on souhaite lire ou modifier.

10.3.3 Les fonctions de gestion de fichiers

- `FILE* fopen(const char* "nomFichier", const char* modeOuverture)`
  Permet d'ouvrir un fichier en lecture ou en écriture selon la valeur de la chaîne constante modeOuverture : "r" pour *lecture*, "w" pour *écriture* (si le fichier existe, il est réinitialisé), "a" pour *ajout* (si le fichier existe, on écrit à la suite). Pour les deux derniers modes, s'il n'existe pas, le fichier spécifié est créé. Les valeurs "rb", "wb" et "ab" seront utilisées lorsque l'on devra ouvrir des fichiers binaires et non des fichiers "texte" comme précédemment. L'option "+" ajoutée au modeOuverture donne la possibilité de faire de la mise à jour (ouverture à la fois en lecture et en écriture). La fonction `fopen` renvoie NULL en cas d'échec, un pointeur sur le fichier en cas de réussite (en début ou fin de fichier selon le mode d'ouverture).

- `int fclose(FILE* f)`
  Permet de fermer un fichier ouvert par la fonction `fopen` ; renvoie 0 en cas de réussite, EOF sinon (et la variable système `errno` est alors renseignée au code erreur correspondant).

- `int fflush(FILE* f)`
  Permet de vider les buffers dans le flux **f** ; si **f** est NULL, vide simplement les buffers sur la sortie standard ; renvoie 0 en cas de réussite, EOF sinon (et la variable système `errno` est alors renseignée).

- La bibliothèque standard offre deux modes d'accès aux fichiers : l'accès séquentiel et l'accès direct. Sans précaution particulière, l'accès est séquentiel. Cela signifie qu'après lecture ou écriture d'un article d'un fichier, on passe à l'article suivant. Cet accès séquentiel est automatiquement réalisé par les fonctions `fputc, fgetc, fgets, fputs, fprintf, fscanf, fread,` ... La fonction `ungetc`, qui n'est pas une fonction de lecture ou d'écriture, permet un déplacement arrière d'un octet.
  Lors de l'accès direct, la fonction `fseek` permet de se positionner directement en un octet quelconque d'un fichier existant. La fonction `ftell` donne la position courante, en octet, dans un fichier, position par rapport au début du fichier. La fonction `rewind` permet de se positionner en début de fichier.

- `int feof(pointeur)`
  Permet de savoir si `pointeur` pointe vers la fin du fichier. Cette fonction retourne la valeur 0 si `pointeur` ne pointe pas vers la fin du fichier et une valeur différente de 0 dans le cas contraire.

10.3.4 Extension des Entrées/Sorties classiques

- Les Entrées/Sorties formatées
  `fprintf(FILE*, char const*, [`$\texttt{arg}_1$`,`$\texttt{arg}_2$`,...,`$\texttt{arg}_n$`])`
  Permet d'écrire dans un fichier en spécifiant le format des éventuels arguments. En réalité, cette fonction est un "cas général" de `printf` :

`printf("chaineFormatee", liste_arguments) ==`
`fprintf(stdout, "chaineFormatee", liste_arguments)`

`fscanf(FILE*, char const*, [&arg`$_1$`,&arg`$_2$`,...,&arg`$_n$`])`
Permet de lire des éléments et de préserver les valeurs lues en spécifiant le format de lecture des arguments. En réalité, cette fonction est un "cas général" de `scanf` :
`scanf("chaineFormateeé", liste_adresses) ==fscanf(stdin, "chaineFormatee", liste_adresses)`

- Les Entrées/Sorties non formatées
  Comme pour le cas classique, nous avons, pour les fichiers, les quatre fonctions d'Entrées/Sorties non formatées suivantes :
  `fgetc(pointeur)`, `fputc(c,pointeur)`, `fgets(pt_chaîne,n,pointeur)`, `fputs(pt_chaîne,pointeur)`.
  A ces fonctions, on peut ajouter la fonction `ungetc(c,pointeur)` qui annule le déplacement occasionné par un précédent `getc`, ce qui signifie que l'on revient en arrière d'un octet. L'argument `c` doit être le dernier caractère lu et ne peut être égal à `EOF`, de plus la fonction `ungetc` ne s'utilise qu'en mode lecture.

10.3.5 Entrées/Sorties binaires générales

Les fonctions `fread` et `fwrite` permettent la lecture et l'écriture, en binaire, de tout type de données, notamment les ensembles de données organisées, tableaux et structures.

- La fonction `fwrite` permet d'écrire dans le fichier concerné n objets successifs ainsi déterminés :
  `fwrite(pointeur_objet,taille_objet,n,pointeur)`
  `pointeur_objet` étant un pointeur vers un objet particulier,
  `taille_objet` étant la taille de cet objet,
  `pointeur` étant le pointeur associé au fichier,
  `n` étant le nombre d'objets successifs à écrire.

- Inversement, la fonction :
  `fread(pointeur_objet,taille_objet,n,pointeur)`
  permet de lire dans le fichier associé à `pointeur` `n` objets dont le type correspond à `pointeur_objet` et la taille est égale à `taille_objet`. Ces `n` objets seront rangés en mémoire à partir de l'adresse représentée par `pointeur_objet`.

10.3.6 L'accès direct

- La fonction `fseek` permet de se positionner sur un octet quelconque d'un fichier, afin de pouvoir lire ou écrire un article à cet endroit précis du fichier. Cette fonction `fseek` s'écrit :
  `fseek(pointeur, déplacement, origine)`
  `pointeur` est le pointeur de type `FILE` attaché au fichier.
  `déplacement` détermine la nouvelle position de `pointeur` dans le fichier. Le déplacement est un entier relatif de type `long` et il est compté en nombres d'octets par rapport à l'origine.
  `origine` prend une des trois valeurs suivantes qui désignent respectivement :

0 : le début du fichier
1 : la position courante du pointeur
2 : la fin du fichier

En cas d'opération incorrecte, la fonction `fseek` qui est du type `int`, retourne une valeur différente de 0.

- La fonction `rewind(pointeur)` permet de se positionner en début de fichier, elle est du type `void` et ne retourne donc aucune valeur.
- La fonction `ftell(pointeur)` donne, en nombre d'octets, la position courante par rapport au début du fichier, elle est du type `long int`.

Bibliographie

P. Dax, *Langage C*, Eyrolles, 1992.

C. Delannoy, *Programmer en langage C : Cours et exercices corrigés*, Eyrolles, 2002.

S. Harbisson, *Langage C*, Eyrolles, 1989.

B.W. Kernighan et D.M. Ritchie, *The C programming language*, Prentice-Hall, Inc. 1978.

B.W. Kernighan et D.M. Ritchie, *Langage C - Norme ANSI*, Eyrolles, 2004.