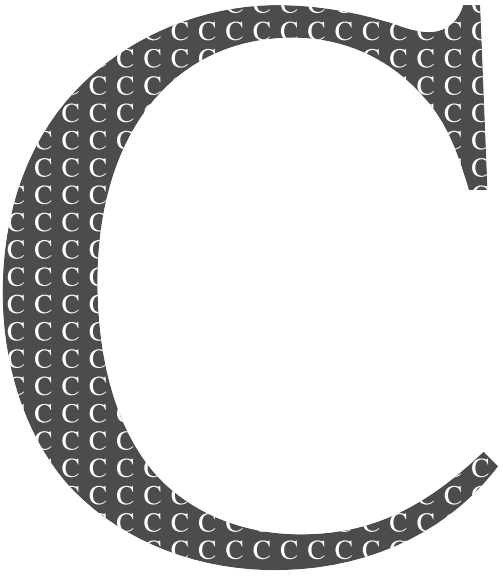
INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

Année scolaire
2016-2017

Langage C



Éric Guérin

5 octobre 2016

TABLE DES MATIÈRES

CHAPITRE 1
LES FONDAMENTAUX

1.1 Premier exemple

Voici un exemple de programme source en C, comportant une seule fonction `main` :

```
#include <stdio.h>
#define TVA 20.6 /* on définit le taux de TVA en constante */
void main(void)
{
     float HT,TTC;
     puts ("veuillez entrer le prix H.T.");
     scanf("%f",&HT);
     TTC=HT*(1+(TVA/100));
     printf("prix T.T.C. %f\n",TTC);
}
```

On trouve dans ce programme les éléments fondamentaux suivants :

— Des directives du préprocesseur (commençant par #)

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

Cette directive permet d'inclure un fichier d'en-tête standard. Le nom du fichier ici est `stdio.h`, et déclare des fonctions d'entrées/sorties (stdio signifie standard input/output) comme `puts`, `scanf` ou `printf` utilisées dans la suite du programme. Ces fonctions permettent au programme d'interagir avec la console (donc le clavier et l'écran).

```
#define TVA 20.6
```

Cette directive permet de définir une macro. Ici, la macro n'a pas d'argument, on parle alors de constante. Ainsi, à chaque fois que le préprocesseur rencontrera le mot `TVA` il le remplacera par sa définition, c'est-à-dire `20.6`. C'est un remplacement « basique » faisant intervenir une analyse très simpliste de la syntaxe du programme source.

— Un en-tête de fonction

```
void main(void)
```

Ici, il n'y a qu'une seule fonction de définie, elle porte le nom de `main`. Cette fonction est obligatoire dans tout programme destiné à être exécuté, car elle fournit un point d'entrée. C'est la fonction qui est appelée au début du programme. Dans le cas où vous écrivez une bibliothèque, ce ne sera pas nécessaire (une bibliothèque est un catalogue de fonctions).

— Un bloc d'instructions délimité par des accolades { et }. Il sert à donner une structure au programme source, transformant plusieurs instructions successives en une instruction de plus haut niveau.
— Des déclarations de variables

```
float HT,TTC;
```

On donne en premier le type puis une liste de variables qui auront ce type. En C, la déclaration de variable est obligatoire. Une variable est une case mémoire que l'on réserve dans laquelle on stocke une valeur. Ici, il s'agit d'une valeur de type pseudo-réel (`float` à cause de la virgule flottante). Il existe trois types scalaires de base en C : l'entier `int`, le flottant `float` et le caractère `char`. On essaye de choisir des noms de variables explicites (plus d'une lettre et tous différents).
— Des instructions qui sont terminées par un point-virgule. Une instruction est un ordre élémentaire que l'on donne à la machine, qui manipulera les données (variables) du programme, ici soit par appel de fonctions (`puts`, `scanf`, `printf`) soit par affectation (=).

Voici le détail des 4 instructions présentes dans ce programme :

```
puts ("veuillez entrer le prix H.T.");
```

permet d'afficher à l'écran (dans la console) la chaîne de caractères passée en argument. Il s'agit d'un appel à la fonction `puts` déclarée dans `stdio.h`.

```
scanf("%f",&HT);
```

permet d'attendre la saisie d'un nombre au clavier au format flottant. L'appel à la fonction `scanf` comprend deux arguments : le format (ici `"%f"` pour signifier flottant) et l'adresse mémoire d'une variable (obtenue grâce à l'opérateur & appliqué à la variable `HT`).

```
TTC=HT*(1+(TVA/100));
```

est une opération d'affectation obtenue grâce au signe = (attention, ce n'est pas l'opérateur d'égalité). Ce qui se trouve à droite du signe est évalué puis affecté à la variable qui se trouve à gauche du signe. Ici, on divise la variable `TVA` par la constante 100, on ajoute à cela la valeur 1, puis on multiplie par la variable `HT`. Le résultat est ensuite mis dans la variable `TTC`.

```
printf("prix T.T.C. %f\n",TTC);
```

affiche le résultat en combinant une chaîne constante avec une variable (ici, `TTC`).

1.2 Les instructions

▶ Notion de données

C'est une notion fondamentale, c'est ce qui représente l'information (n'oubliez pas que l'on parle d'informatique). Les sources de ces données peuvent être de différentes natures :
— Textuelles
— Numériques
— Graphiques
— Sonores
— *etc.*

On parle aussi de flux de données (*stream* en anglais) pour exprimer le fait que ces données transitent de ou vers l'ordinateur.

Donnée en entrée Il s'agit d'une donnée qui vient de l'extérieur de l'ordinateur vers celui-ci. Exemples : une touche tapée au clavier, un clic de souris, le scan d'une image, *etc.*

Donnée en sortie Il s'agit d'une donnée qui va de l'ordinateur vers l'extérieur de celui-ci. Exemples : affichage d'un caractère à l'écran, impression d'un point sur une imprimante, émission d'un son sur un haut parleur, *etc.*

En C (comme sous unix), les échanges en entrée et en sortie de type caractères se traduisent par deux flux :

L'entrée standard ce sont les caractères tapés au clavier dans la console qui a lancé le programme (*standard input* en anglais)

La sortie standard ce sont les caractères qui sont affichés dans la console qui a lancé le programme (*standard output* en anglais)

⚠ Ces flux peuvent être *redirigés* de telle manière qu'ils ne correspondent plus à des interactions avec l'utilisateur (exemple : la sortie standard est redirigée vers un fichier)

▶ Rôle du programme

Le rôle du programme va être de manipuler les données pour en créer d'autres. Typiquement, on utilise les données d'entrée pour calculer des données en sortie (voir le premier exemple de programme). Pour manipuler ces données, on utilise des instructions. Il en existe trois types :

— Les instructions simples
— Les instructions composées
— Les instructions de contrôle

▶ Instructions simples

Une instruction simple permet d'effectuer une opération élémentaire comme l'appel d'une fonction, une affectation, ou les deux... Exemples :

```
a = b*2 + 1;
a = fabs(b);
```

⚠ La fin d'une instruction simple est indiquée par le point-virgule et non la fin de la ligne, on peut donc mettre plusieurs instructions simples sur une même ligne, elles seront exécutées l'une après l'autre. Exemple :

```
fscanf("%d",&b); b++;
```

▶ Instructions composées

Une instruction composée est un ensemble d'instructions (de nature quelconque) regroupées dans un bloc grâce à des accolades { et }.

⚠ On ne met pas de point-virgule à la fin d'une instruction composée. Pour vous en souvenir, dites-vous que le compilateur n'a pas besoin de voir de point-virgule pour savoir que c'est la fin, puisqu'il y a déjà une accolade fermante.

Exemple :

```
rep = 'o';
while ((rep <> 'N')&&(rep <> 'n')) /* instruction de contrôle */
{  /* début de l'instruction composée */
   printf("Entrez un nombre positif inférieur a 100 : ");
   scanf("%d",&n); /* ici, on devrait ajouter des calculs... */
   printf("\nVoulez-vous rejouer (O/N)?");
   scanf("%c", &rep);
}  /* fin de l'instruction composée */
```

L'indentation permet de rendre le code plus lisible en alignant verticalement les instructions d'un même bloc et en décalant sur la droite les instructions d'un sous-bloc.

Elle n'est pas obligatoire et d'ailleurs est ignorée (ce qui n'est pas le cas pour d'autres langages comme Python). Il faut faire très attention :

- À utiliser l'indentation lorsque l'on rentre dans un sous-bloc
- À ne pas l'utiliser lorsqu'elle est fausse et induit une lecture erronée. Exemple :

```
if (toto == 1);
    Dessine(toto);
```

Le if se termine par un point-virgule donc n'implique pas de sous-bloc à la suite (le sous-bloc est alors vide, ce test ne sert à rien). La deuxième ligne sera exécutée quel que soit le résultat du test. L'indentation fait croire le contraire.

⚠ C'est une faute de débutant (voire même de moins débutant) très courante

▶ Instructions de contrôle

Elle servent à traduire l'algorithmique du programme pour faire :

- Des boucles (avec `for`, `while` et `do`)
- Des tests (avec `if` et `else`)
- Des tests multiples (avec `switch .. case`)

Chaque structure a sa propre construction mêlant en général une expression booléenne à l'intérieur de parenthèses avec un ou plusieurs blocs d'instructions.
Voici quelques exemples qui vont seront utiles rapidement pour programmer :

```
/* une boucle de 1 à 10 */
int i;
for (i=1;i<=10;i++) {
    ...
}

/* un test pour savoir si a est positif */
if (a>0) {
    ...
}

/* une boucle qui continue tant que val est négatif */
while (val<0) {
    ...
}
```

1.3 Les fonctions

- Le fonctionnement de base du C repose sur les fonctions.
- La seule obligatoire pour un programme (mais pas une bibliothèque) est la fonction `main`.
- Une fonction renferme en général un algorithme de base qui ne doit pas dépasser en théorie une page.
- Multiplier le nombre de fonctions peut être un but de programmation mais il faut garder en vue qu'un trop grand nombre d'appels de fonctions ralentit l'exécution du programme (on pourra alors recourir aux macros).
- Une fonction communique avec l'extérieur par l'intermédiaire de ses paramètres et de sa valeur de retour.

▶ Prototype d'une fonction

Le *prototype* d'une fonction (ou déclaration) est la donnée du type de ses arguments, du type de sa valeur de retour, ainsi que de son nom :

```
type_retour nom_fonction (type_paramètre1 nom_paramètre1,
                          type_paramètre2 nom_paramètre2,
                          ...);
```

Comme toutes les déclarations en C, on met un point-virgule à la fin.

⚠ Dans la déclaration, les noms des paramètres sont facultatifs, ce qui n'est pas le cas pour la définition de cette même fonction

▶ Définition de la fonction

À chaque déclaration de fonction, on associe une définition, qui correspond au contenu de celle-ci, c'est-à-dire au code source associé :

```
type_retour nom_fonction (type_paramètre1 nom_paramètre1,
                          type_paramètre2 nom_paramètre2,
                          ...)
{
  /* déclarations de variables locales */
  /* instructions de la fonction */
  return (...);
}
```

On a donc simplement enlevé le point-virgule de la déclaration, et ajouté un bloc d'instructions (avec des déclarations) qui se termine par un `return` qui indique l'expression qui doit être retournée à l'appelant.

▶ Appel d'une fonction

Contrairement à la déclaration et la définition qui doivent être en exemplaire unique, l'appel de la fonction peut être fait plusieurs fois. Il consiste à utiliser la fonction que l'on a définie avec des arguments :

```
variable = nom_fonction(valeur1, valeur2, ...);
```

C'est une instruction simple qui se termine donc par un point-virgule.

- À cet endroit précis du programme, le contexte change, et on bascule dans l'exécution de la fonction.
- Les valeurs données en argument de l'appel (dans l'exemple, il s'agit de `valeur1` et `valeur2`) sont communiquées au nouveau contexte de la fonction (sous la forme des variables `nom_paramètre1` et `nom_paramètre2`).
- À l'issue de l'exécution de la fonction, la valeur exprimée dans le `return` est retournée au contexte appelant, comme si `nom_fonction` prenait cette valeur.

▶ Exemple complet

- Programme complet faisant la moyenne de deux flottants
- Une fonction `moyenne_de_deux_float` qui prend deux `float` en argument et renvoie un `float` comme résultat
- La fonction `main` utilise cette nouvelle fonction qui doit être déclarée avant l'appel
- La définition de la fonction `moyenne_de_deux_float` peut être faite seulement après la fonction `main`

```c
#include <stdio.h>

/* prototype de la fonction */
float moyenne_de_deux_float(float a,float b);

void main(void)
{
   float x,y;
   float moyenne;
   puts("Entrez deux flottants svp");
   scanf("%f",&x);
   scanf("%f",&y);
   /* appel de la fonction */
   moyenne = moyenne_de_deux_float(x,y);
   printf("La moyenne de %f et %f est %f\n",x,y,moyenne);
}

/* définition de la fonction */
float moyenne_de_deux_float(float a,float b)
{
   return (0.5*(a+b));
}
```

À l'exécution, cela donne :

```
$ ./exfun
Entrez deux flottants svp
1.2
2.6
La moyenne de 1.200000 et 2.600000 est 1.900000
$
```

▶ Modification de la valeur d'un paramètre

En C, tous les passages de paramètres se font par valeur. Cela signifie que lorsque l'on modifie dans la fonction un des paramètres, il n'est pas modifié dans le contexte appelant (l'endroit où l'on a fait l'appel à la fonction). C'est l'équivalent du mot clé val utilisé en algorithmique. Exemple :

```c
void fonction1(int valeur)
{
   valeur = 1;
}

...

int a;
a=0;
fonction1(a);
printf("%d\n",a); /* affiche 0 */
```

Pour pouvoir modifier un paramètre, il faut passer explicitement l'adresse de la variable.
Pour extraire l'adresse de la variable, on utilise l'opérateur &, pour obtenir la variable à partir de son adresse, on utilise l'opérateur *. Cela donne :

```c
void fonction2(int * valeur)
```

```
{
    *valeur = 1; /* l'opérateur * permet de retrouver la
                    variable associée */
}

...

int a;
a=0;
fonction2(&a); /* l'opérateur & permet d'extraire l'adresse
                  de la variable */
printf("%d\n",a); /* affiche 1 */
```

▶ Remarques en vrac

— Dans la mesure du possible, on évitera de faire des entrées/sorties dans des fonctions, sauf si c'est leur but (une fonction qui fait uniquement de l'affichage par exemple), cela permet plus de modularité. Les fonctions de calculs ne doivent pas être liées à l'utilisation d'une console, on peut très bien imaginer les utiliser dans un environnement graphique où les interactions seront gérées complètement différemment.

— Lorsque le code d'une fonction est très court comme dans l'exemple précédent, on utilise souvent une macro pour accélérer les calculs. On aurait par exemple pu écrire :

```
#define moyenne_de_deux_nombres(a,b) (0.5*(a+b))
```

et éviter ainsi les passages d'arguments dans la pile.

— On aurait pu éviter la déclaration de la variable `moyenne`, car elle n'est utilisée qu'une fois par la suite de son affectation. Pour cela, il aurait fallu écrire directement :

```
printf("La moyenne de %f et %f est %f\n",x,y,
                    moyenne_de_deux_float(x,y));
```

— L'intérêt de séparer prototype et définition ne doit pas vous sauter aux yeux. Pourtant, c'est primordial, cela permet en outre de les mettre dans deux fichiers différents :
  — un fichier d'en-tête (`.h`) contenant tous les prototypes,
  — un fichier d'implémentation (`.c`) contenant les définitions.

— `void` signifie *vide*, il est utilisé comme type pour indiquer qu'il n'y en a pas (pas d'argument dans la fonction par exemple, ou pas de valeur retour de la fonction)

▶ Fonctions/procédures

En C, les procédures n'existent pas vraiment. Tout est fonction. Pour indiquer qu'il s'agit d'une fonction ne renvoyant aucune valeur retour, on déclare comme type de retour `void`. Lors de l'appel, on ne peut pas récupérer de valeur de retour, donc l'utilisation dans une expression sera une erreur.

```
int toto(int parametre);
void tata(int parametre);
int a;
a = toto(12);
tata(13);
a = tata(13); /* est faux */
```

et provoquera ce genre de message d'erreur :

```
faux.c:7: error: void value not ignored as it ought to be
```

que l'on peut traduire en « Erreur : valeur vide non ignorée comme elle devrait l'être ».

▶ Return

Attention, ce mot-clé a une double signification :
- Il renvoie le résultat au contexte appelant
- Il redonne le contrôle au contexte appelant

Cela signifie que de code placé derrière ne sera jamais éxécuté. Il se peut cependant que du code se trouve après dans le cas de tests. Exemple :

```
int division_entiere_sans_erreur(int a, int b)
{
  int resultat;
  if (b == 0)
    return 0;
  resultat = a/b;
  return resultat;
}
```

est une fonction qui fait la division entière avec un test de validité du dénominateur. On trouve deux fois le mot-clé `return`, pourtant à chaque exécution, un seul sera exécuté. La règle à suivre : il faut que dans tous les cas de figure (*i.e.* tous les cas de tests), une valeur soit renvoyée.
Exemple de fonction non valide :

```
int fonction_ne_renvoyant_pas_toujours_une_valeur(int a)
{
  if (a>0)
    return a+2;
  if (a<0)
    return a-1;
}
```

Dans le cas où `a` est nul, la fonction ne renvoie pas de valeur. Le compilateur est parfois capable de détecter ce genre de problème quand les messages d'avertissements sont activés, et affichera ce type de message (ce n'est pas une erreur) :

```
$ gcc -Wall -c -o nonretour.o nonretour.c
nonretour.c: In function 'fonction_ne_renvoyant_pas_toujours_une_valeur':
nonretour.c:7: warning: control reaches end of non-void function
```

qui signifie « Avertissement : on a atteint la fin d'une fonction renvoyant une valeur ». Or on ne devrait jamais atteindre cette fin (la dernière accolade) dans l'exécution du programme, puisqu'un `return` doit l'interrompre.

- ⚠ `return` ne fait en aucun cas un affichage
- ⚠ `return` n'est pas une fonction, donc les parenthèses ne sont pas obligatoires, les deux lignes suivantes sont équivalentes :

```
return a*0.5;
return (a*0.5);
```

1.4 Les types de données

▶ Types de base

Les types de base du langage C sont les suivants :
- `int` : entier

— `float`, `double` : réel. En fait, il ne s'agit pas d'un réel mais plutôt d'un nombre à virgule flottante dont la précision est limitée (un peu moins pour le type `double`).
— `char` : un caractère (*character* en anglais)
— `void` : vide (pas de type ou type quelconque)

▶ Modificateurs

On peut modifier les types de base en plaçant devant eux des modificateurs qui peuvent être de différentes natures :
— Signe
  — `signed` indique que le nombre peut être négatif (par défaut)
  — `unsigned` indique que le nombre est uniquement positif
— Longueur de codage
  — `short` pour diminuer la longueur de codage
  — `long` pour augmenter la longueur de codage
— Lecture seule
  — `const` indique au compilateur qu'il ne peut y avoir aucune modification sur la variable associée (c'est donc une constante)

Certaines combinaisons de modificateurs et de types sont impossibles ou n'auront aucun effet. Exemple : `long char` n'a pas vraiment de sens car un caractère est toujours codé sur un seul octet.

1.5 Codage (représentation)

La notion de codage fait intervenir la façon dont est représenté la variable dans la mémoire de l'ordinateur, c'est-à-dire combien de bits sont utilisés et quelle est la signification de chacun.

▶ Codage des entiers

Pour les entiers (`int` mais aussi `char`), c'est simple. Tous les bits ont la même signification sauf éventuellement le premier qui indique le signe lorsqu'il y en a un. Exemple pour un `unsigned char` :

Numéro du bit	7	6	5	4	3	2	1	0
Puissances de 2	128	64	32	16	8	4	2	1
Exemple de valeur	1	0	1	1	0	0	1	1
Valeur en décimal	128	0	32	16	0	0	2	1

La somme est 179. Ainsi, une variable du type `unsigned char` qui aura la valeur 179 sera représentée en mémoire par un octet dont les bits seront 10110011.
Pour les `int` et toutes les autres variantes, le principe est le même mais avec plus de bits.

Pour le codage des nombre négatifs, on utilise la complémentation à 2 qui ne sera pas détaillée dans ce cours, retenez seulement que le bit de poids fort indique le signe mais que l'on ne calcule pas l'opposée en changeant simplement ce bit.

▶ Codage des flottants

Le codage des flottants est un peu plus complexe car il fait intervenir trois groupes de bits :
— Le signe (un seul bit)
— L'exposant
— La mantisse

La notion de virgule flottante fait référence au fait que l'on va coder un nombre comme 0.000002004 sous la forme 0.2004×10^{-5}. Le nombre 0.2004 est appelé la mantisse et -5 est l'exposant. Ceci résume à peu près le codage au détail près que tout est en binaire (ou même hexadécimal). Exemple : le codage du nombre 18.75 sur 32 bits.

$$\begin{aligned} 18.75 &= 1 \times 16^1 + 2 \times 16^0 + 12 \times 16^{-1} \\ &= 16^{\mathbf{2}} \ (\mathbf{1} \times 16^{-1} + \mathbf{2} \times 16^{-2} + \mathbf{12} \times 16^{-3}) \end{aligned}$$

La mantisse en hexadécimal est 12C soit 12 Co 00 sur 3 octets (les zéros qui suivent peuvent servir à ajouter des décimales de précision).
L'exposant est égal à 2 : on le code en lui ajoutant 64 : $64 + 2 = 66$ soit 42 en hexadécimal.

Le codage final sera donc :

Signe	Exposant		Mantisse					
+	4	2	1	2	C	0	0	0
0	100	0010	0001	0010	1100	0000	0000	0000

▶ Correspondance ASCII

Comme nous l'avons vu précédemment, un caractère est codé par une variable de type `char` qui n'est rien d'autre qu'un entier sur 8 bits. La correspondance entre cet entier et le caractère se fait grâce à la table ASCII[1] :

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

▶ Comment lire cette table ?

— On obtient le code d'un caractère en lisant le code hexadécimal de la ligne puis celui de la colonne et en les concaténant. Exemple : la parenthèse ouvrante correspond à la ligne 2 et la colonne 8 soit 0x28. En décimal, cela correspond à $16 \times 2 + 8 = 40$.
— On obtient le caractère correspondant à un code en cherchant celui qui correspond à la bonne ligne et la bonne colonne. Exemple, je cherche le caractère qui correspond au code 122 soit 0x7A. À la ligne 7 et la colonne A, je lis le caractère z.

▶ Que contient cette table ?

— Les deux premières lignes correspondent à des caractères spéciaux de contrôle. Les plus connus sont :
  — Le signal sonore BEL (bell) au code 0x07 (soit 7 en décimal)
  — La tabulation horizontale TAB au code 0x09 (soit 9 en décimal)
  — Le saut de ligne LF (line feed) au code 0x0A (soit 10 en décimal)
  — Le retour chariot CR (carriage return) au code 0x0D (soit 13 en décimal)

1. American Standard Code for Information Interchange

  - Le caractère d'échappement ESC (escape) au code 0x1B (soit 27 en décimal)
- À partir de la troisième ligne commencent les caractères imprimables
  - Les chiffres vont de 0x30 à 0x39 (soit 48 à 58 en décimal)
  - Les lettres majuscules vont de 0x41 à 0x5A (soit 65 à 90 en décimal)
  - Les lettres minuscules vont de 0x61 à 0x7A (soit 97 à 122 en décimal)

▶ Pourquoi seulement 128 caractères ?

- Ce sont les seuls qui sont vraiment standards au niveau ASCII, ensuite cela dépend de la plateforme.
- Il existe beaucoup de standards
- Un des plus connus est le standard ISO-8859-1 ou latin-1. Il définit les caractères accentués minuscules et majuscules.
- Certains codages de caractères utilisent plusieurs caractères pour coder une seule lettre, c'est le cas de l'encodage UTF-8 de plus en plus souvent utilisé.

▶ Limites de codage

Que ce soit pour les entiers ou pour les flottants, la précision des nombres n'est pas infinie. On ne peut donc pas considérer que les calculs sont exacts comme dans une formule mathématique. Les erreurs suivantes peuvent survenir :

Dépassement de capacité. Pour les entiers, cela consiste à dépasser la valeur minimum ou maximum, les effets peuvent être catastrophiques.

Erreur d'arrondi. En C, `1.0 + 1.0` n'est pas forcément égal à `2.0`. Ceci est du au codage qui approxime la mantisse par une décomposition binaire (et non décimale) finie (voir le paragraphe codage des flottants).

Erreur d'arithmétique. Le nombre est tellement petit qu'il devient égal à 0. On peut avoir le même effet pour les nombres très grands.

Erreur d'annulation. Si on additionne deux nombres ayant des proportions très différentes, on peut dépasser la précision de la mantisse et le résultat sera le plus grand des deux nombres. Exemple : `100000.0 + 0.000000000001` sera égal à `100000.0`.

Il existe des bibliothèques qui gèrent le calcul en précision arbitraire, mais elles sont à utiliser qu'en cas d'extrême nécessité. Le plus souvent, des erreurs dues à des limites de codage peuvent être évitées facilement. Par exemple, il faut à tout pris éviter ce genre de choses :

```
float x;
for (x=0.0 ; x<=10.0 ; x+=0.1) { ... }
```

et préférer :

```
int i;
float x;
for (i=0 ; i<=100 ; i++) {
  x = (float) i / 10.0;
  ...
}
```

Dans le deuxième cas, on est sûr du nombre d'itérations.

1.6 Les constantes

Les constantes sont des valeurs fixes dans un programme. Elles ne bougent pas. Il en existe trois types :
- Constante numérique.
- Constante caractère.
- Constante chaîne de caractères.

▶ Constantes numériques

Les constantes entières peuvent être données en décimal, en octal (base 8) et en hexadécimal.

Décimal. On ne met pas de 0 devant :

```
1   23   9999
```

Octal. On met un 0 devant :

```
01  027  023417
```

Hexadécimal. On met 0x devant :

```
0x1 0x17 0x270F
```

Pour spécifier le codage utilisé pour la constante, on ajoute des suffixes :

```
long int nl;
nl = 240L;
unsigned int iu;
ui = 701u;
```

Pour afficher avec un certain format avec `printf` on utilise `%d`, `%o` et `%x` pour le décimal, l'octal et l'hexadécimal. Exemple :

```
printf("%d %o %x\n",23,23,23);
```

donne à l'exécution :

```
23 27 17
```

▶ Constantes caractères

En C, nous avons déjà vu qu'un caractère est en fait un entier. Si l'on veut directement faire référence à une constante de type caractère, on utilise les guillemets simples. Exemples :

```
'a' 'b' ... 'z' 'A' 'B' ... 'Z' '0' ... '9'
```

mais il existe aussi des constantes pour certains caractères spéciaux :
- '\n' le retour chariot
- '\t' la tabulation
- '\'' l'apostrophe
- *etc.*

▶ Constantes chaînes de caractères

Il n'existe pas en C de type prédéfini pour les chaînes de caractères (string en anglais), on est obligé de faire un tableau.

```
char chaine[10];
```

est une chaîne de caractères de longueur 10.

⚠ Il faut toujours ajouter 1 à la longueur d'une chaîne car elle se termine par un '\0'. Ainsi, dans `chaine` on ne pourra mettre que 9 caractères effectifs.

Il est possible de définir des constantes chaînes directement avec les guillemets doubles. Exemples :

```
"Bonjour\n"
"16/09/2004"
```

Dans ce cas là, la taille du tableau est automatiquement calculée et l'allocation est faite.

⚠ On ne peut pas modifier les chaînes de caractères constantes pour plusieurs raisons :
- Les données sont directement mises dans le code exécutable du programme par le compilateur. On s'interdit de modifier dynamiquement le code du programme.
- La taille ne peut pas augmenter car elle est fixée à l'avance.

▶ Constantes symboliques

Lorsqu'une constante est utilisée plusieurs fois dans un programme, on peut éviter de la retaper à chaque fois que l'on veut l'utiliser en déclarant une *constante symbolique* grâce à la directive de compilation `#define`. Dans ce cas là, c'est le préprocesseur qui transforme le code avant la compilation. Exemples :

```
#define MAX 1000
#define TVA 19.6
#define VRAI 1
#define FAUX 0
#define MESSAGE_ERREUR "Erreur dans le programme"
```

⚠ Ne surtout pas mettre de point virgule à la fin de la ligne

Cette construction a l'avantage d'éviter de déclarer une variable et de pouvoir modifier le source une seule fois pour toutes les occurrences.

1.7 Conventions d'écriture

L'ordre dans lequel on écrit le contenu d'un programme source doit être respecté pour plus de lisibilité.

▶ Pour un programme

- Les directives `#include`
- Les directives `#define`
- Les définitions de variables globales (à éviter)
- Les prototypes de fonctions
- `int main(...)`
- `{ ... }`
- Définition des autres fonctions (celles dont le prototype a été donné plus haut et éventuellement d'autres fonctions qu'elles utilisent mais pas d'imbrication)

▶ Pour une fonction

- En-tête de fonction `type_retour nom_fonction(type1 parametre1, ...)`
- `{`
- Déclaration de variables locales
- Saut de ligne
- Corps de la fonction (liste d'instructions)
- Saut de ligne

— `return valeur;`
— `}`

▶ Noms de variables

Un identificateur peut contenir les caractères suivants :
— Les lettres majuscules et minuscules
— Les chiffres mais pas en première position
— Le blanc souligné `_` mais pas en première position car il est réservé au compilateur

Exemples :

✘ `_identificateur` ✔ `identificateur`
✘ `élément` ✔ `element`
✘ `1ere_Valeur` ✔ `Valeur1`

On utilise en général des noms de variables ou de constantes (*identificateurs*) qui sont représentatifs de leur contenu. Par exemple :
— `TVA` pour un taux de TVA
— `moyenne` ou éventuellement `moy` pour une moyenne mais mieux vaut éviter `m`
— `i`, `j`, `k` sont tolérés pour des compteurs de boucles

De même on utilise des majuscules et les minuscules pour différencier les constantes des variables globales ainsi que locales :
— Une variable locale est entièrement en minuscule
— Une variable globale commence par une majuscule de même qu'un nouveau nom de type
— Une constante est entièrement en majuscule
C'est ainsi beaucoup plus facile de s'y retrouver.

1.8 Valeurs logiques

En C, le type de données booléen n'existe pas. Il s'agit du type entier auquel on fait correspondre les valeurs vrai et faux :
— 0 signifie faux
— tout le reste est vrai (y compris des valeurs négatives)

Ainsi les deux tests suivants sont complètement équivalents :

```
if (x)
if (x != 0)
```

Un test peut donc être n'importe quelle expression numérique renvoyant un entier. Si le résultat est nul, c'est équivalent à une expression booléenne fausse, sinon c'est une expression booléenne vraie.

Questions de synthèse

1. L'opérateur = est équivalent en algorithmique à la flèche à gauche d'affectation, vrai ou faux ?
2. Une instruction composée peut contenir des instructions simples comme des instructions composées ou de contrôle, vrai ou faux ?
3. Comment délimite-t-on une instruction composée ?
4. De manière générale, une macro est plus lente qu'une fonction, vrai ou faux ?
5. Les procédures en C n'existent pas vraiment, vrai ou faux ?
6. `return` est-t-il un mot-clé du C ou bien une fonction ?
7. Une fonction ne renvoyant pas `void` doit impérativement toujours renvoyer une valeur, vrai ou faux ?
8. Par défaut un entier `int` est-t-il signé ou non signé ?
9. Un `float` est-t-il plutôt un réel ou un nombre à virgule ?
10. Quelles sont les trois parties qui composent le codage d'un flottant ?
11. En C, écrire `'A'` revient à écrire le nombre `65`, vrai ou faux ?
12. Combien la table ASCII standard définit-t-elle de caractères ?
13. Les caractères de la table ASCII sont-t-ils tous imprimables ?
14. Peut-t-on utiliser les accents dans les noms de variables ? Les tirets - ? Les blancs soulignés _ ? Les chiffres ?
15. Qu'est-ce qui est considéré comme booléen vrai en C ?

Réponses : 1. Vrai. 2. Vrai. 3. Avec des accolades. 4. Faux, elle est plus rapide car ne nécessite pas de passage d'argument dans la pile. 5. Vrai. 6. C'est un mot-clé. 7. Vrai, dans tous les cas de figure, elle doit renvoyer une valeur. 8. Il est signé (signed). 9. C'est plutôt un nombre à virgule. 10. Signe, mantisse et exposant. 11. Vrai. 12. 128 caractères. 13. Non, en dessous de 32, ils ne sont pas imprimables, ce sont des caractères de contrôle. 14. Pas d'accents, pas de tirets, on peut utiliser les blancs soulignés mais pas au début car cela est réservé au compilateur, on peut utiliser les chiffres mais pas au début. Tout entier différent de la valeur 0.

CHAPITRE 2
LES STRUCTURES DE CONTRÔLE

2.1 while, do while

while permet de faire une boucle « tant que » avec le test en début tandis que `do { } while` permet de faire la même boucle mais avec le test à la fin.

▶ while

Algorithmique	En C	Autre présentation

tant que *condition* faire
  ...
└
	`while (condition)`
	`{`
	`  ...`
	`}`
		`while (condition) {`
		`  ...`
		`}`

▶ do while

Algorithmique	En C	Autre présentation

répéter
  ...
jusqu'à *(condition)*;
	`do`
	`{`
	`  ...`
	`} while (!condition);`
		`do {`
		`  ...`
		`} while (!condition);`

▶ Exemples

Un bout de programme qui tourne en boucle tant que l'utilisateur ne saisit pas le nombre 1.

```
#include <stdio.h>

int main(void) {
   int val = 0;

   while (val != 1){
      printf("Tapez un nombre"
          "(1 pour arreter le programme) :\n");
      scanf(" %d",&val);
      printf("Vous avez saisi %d.\n",val);
   }
```

```
    return 0;
}
```

Une autre version mais avec le test à la fin :

```
#include <stdio.h>

int main(void) {
    int val;

    do {
        printf("Tapez un nombre"
            "(1 pour arreter le programme) :\n");
        scanf(" %d",&val);
        printf("Vous avez saisi %d.\n",val);
    } while (val != 1);
    return 0;
}
```

⚠ Ce programme comporte une faille due à la fonction scanf

La version la plus correcte serait donc :

```
#include <stdio.h>

int main(void) {
    int val,ok;

    do {
        printf("Tapez un nombre"
            "(1 pour arreter le programme) :\n");
        ok = scanf(" %d",&val);
        if (ok)
            printf("Vous avez saisi %d.\n",val);
    } while (ok && val != 1);
    return 0;
}
```

2.2 if, else

if permet de faire un « si » et else de rajouter une clause « sinon ».

▶ if simple

Algorithmique	En C	Autre présentation

si *condition* alors
  ...
	`if (condition)`
	`{`
	`  ...`
	`}`
		`if (condition) {`
		`  ...`
		`}`

⚠ La suppression des accolades peut être faite dans le cas où il n'y a qu'une seule instruction dans la clause du if, mais vous devez essayer de l'éviter car elle est source d'erreurs (lorsqu'on rajoute une instruction par exemple).

```
if (condition)
    instruction;
```

- if avec else

Algorithmique	En C	Autre présentation

si *condition* alors
  ...
sinon
  ...
	`if (condition)`
	`{`
	`  ...`
	`}`
	`else`
	`{`
	`  ...`
	`}`
		`if (condition) {`
		`  ...`
		`}`
		`else {`
		`  ...`
		`}`

- if-else imbriqués

On peut enchaîner avec un if la sortie du else lorsque l'on veut raffiner successivement les tests.

Les tests sont effectués chacun leur tour et s'ils ne sont pas satisfait, on passe au test suivant. Si aucun test n'est satisfait, alors c'est la dernière clause else qui est exécutée. Ce dernier else est facultatif (donc si on ne le met pas, il est possible que rien ne soit exécuté).

Algorithmique	En C

si *condition1* alors
  ...
sinon si *condition2* alors
  ...
sinon si *condition3* alors
  ...
sinon
  ...
	`if (condition1) {`
	`  ...`
	`}`
	`else if (condition2) {`
	`  ...`
	`}`
	`else if (condition3) {`
	`  ...`
	`  ...`
	`} else {`
	`  ...`
	`}`

- Exemples

Une fonction qui calcule le maximum de deux nombres entiers et renvoie le résultat.

```
int maximum(int a,int b) {
  int resultat;
  if (a<b)
    resultat = b;
  else
    resultat = a;
  return resultat;
}
```

On peut éviter la clause else en faisant comme ceci :

```
int maximum(int a,int b) {
  int resultat = a;
  if (a<b)
    resultat = b;
  return resultat;
}
```

On peut encore faire plus simple en supprimant la variable locale `resultat` :

```
int maximum(int a,int b) {
  if (a<b)
    return b;
  return a;
}
```

2.3 for

La construction est la suivante :

```
for (instruction_init ; condition ; instruction_increment)
{
  ...
}
```

Le `for` prend donc trois arguments à l'intérieur d'une parenthèse et séparés par des points-virgules :

- Une instruction d'initialisation. Cette instruction est exécutée avant toute chose. C'est presque équivalent à la mettre avant le `for`. En général on initialise un compteur avec cette instruction. Exemple `i=0`.
- Une condition pour continuer la boucle. En général on indique qu'un compteur doit être inférieur à une certaine valeur comme par exemple `i<100`.
- Une instruction qui est exécutée à chaque fin de boucle, juste avant de faire le test pour continuer. En général, on incrémente le compteur car ce n'est pas automatique.

Techniquement, on peut toujours remplacer une boucle `for` par un `while` :

```
instruction_init;
while (condition) {
  ...
  instruction_increment;
}
```

Cependant, on utilisera le `for` pour traduire l'algorithmique du pour :

Algorithmique	En C

pour *(init; condition; iteration)* faire
  ...
	`for (init;condition;iteration)`
	`{`
	`  ...`
	`}`

▶ Exemple simple

Une boucle qui affiche des entiers et leur carré :

```
for (i=1;i<=10;i++)
{
  printf("%d -> %d\n",i,i*i);
}
```

▶ Exemple plus complexe

Une boucle avec un indice croissant et l'autre décroissant pour inverser l'ordre des éléments d'un tableau :

```
for ((croissant=0, decroissant=max);
      croissant <= (max-1)/2;
      (croissant++, decroissant--))
{
  temp = tab[croissant];
  tab[croissant] = tab[decroissant];
  tab[decroissant] = temp;
}
```

▶ Boucles imbriquées

On peut mettre une boucle à l'intérieur d'une autre. Dans ce cas là, il faudra bien veiller à utiliser des identificateurs de compteurs différents. Exemple, affichage d'une table de multiplication :

```
for (i=1;i<=10;i++)
{
  for (j=1;j<=10;j++)
  {
    printf("%4d",i*j);
  }
  printf("\n");
}
```

```
 1   2   3   4   5   6   7   8   9  10
 2   4   6   8  10  12  14  16  18  20
 3   6   9  12  15  18  21  24  27  30
 4   8  12  16  20  24  28  32  36  40
 5  10  15  20  25  30  35  40  45  50
 6  12  18  24  30  36  42  48  54  60
 7  14  21  28  35  42  49  56  63  70
 8  16  24  32  40  48  56  64  72  80
 9  18  27  36  45  54  63  72  81  90
10  20  30  40  50  60  70  80  90 100
```

2.4 switch case

Quand une variable ou une expression peut prendre un certain nombre de valeurs (plus grand que 2), et que pour chacune de ces valeurs on veut faire une action associée, on a la possibilité d'utiliser le `switch case`.

Algorithmique	En C

suivant *expression* faire
  cas où *valeur*1
  ...
  cas où *valeur*2
  ...
  cas où *valeur*3
  ...
  autres cas
  ...
	`switch (expression)`
	`{`
	`  case valeur1:`
	`  ...`
	`  case valeur2:`
	`  ...`
	`  case valeur3:`
	`  ...`
	`  default:`
	`  ...`
	`}`

▶ break

⚠ Si une des valeurs correspond à l'expression, alors toutes les étiquettes seront validées. Si l'on veux empêcher cela, il faut ajouter le mot-clé `break` :

```
case valeur_n:
  ...
  break;
case ...
```

▶ Expressions à utiliser

On peut mettre toutes sortes d'expressions pour des `switch case` mais il faut absolument que ce soit des types énumérés. Sont donc exclus tous les flottants. On peut donc utiliser des entiers (sous toutes leurs formes) ainsi que des caractères. Ceci est donc totalement interdit :

```
switch (cos(angle)) { /* faux */
  case 0.0 :          /* faux */
    printf("Les droites sont perpendiculaires\n");
    break;
  case 1.0 :          /* faux */
    printf("Les droites sont parallèles\n");
    break;
}
```

Par contre on pourra faire ce genre de choses :

```
char note;
...
switch (note) {
  case 'A':
  case 'B':
  case 'C':
    printf("L'examen est validé\n");
    break;
  case 'D':
  case 'E':
    printf("Rattrapage\n");
    break;
  case 'F':
    printf("Redouble\n");
    break;
  default:
    printf("Note invalide\n");
}
```

▶ Exemple : un menu

On utilise très souvent le `switch` pour faire des menus. Il offre l'avantage de les rendre lisibles.

```
int choix,ok;
do {
  printf("Au menu :\n"
      "1 Quitter\n"
      "2 Faire une addition\n"
      "3 Faire une multiplication\n"
      "4 Faire une division\n");
  ok = scanf(" %d",&choix);
  if (ok)
    switch (choix) {
      case 2:
        ...
        break;
      case 3:
        ...
        break;
```

```
        case 4:
          ...
    }
} while (choix != 1 && ok);
```

2.5 continue

L'instruction `continue` est rarement utilisée, elle sert à ne pas exécuter les instructions qui suivent et passer directement à l'itération suivante. Exemple de programmes équivalents :

```
for (i=1;i<=10;i++)
{
  if (i%2)
  {
    printf("%d est impair\n",i);
    continue;
  }
  printf("%d est pair\n",i);
}
```

```
for (i=1;i<=10;i++)
{
  if (i%2)
  {
    printf("%d est impair\n",i);
  }
  else
  {
    printf("%d est pair\n",i);
  }
}
```

La deuxième version est plus claire visuellement.

2.6 goto

L'instruction `goto` permet de faire un saut absolu dans le programme. C'est une aberration algorithmique puisqu'on peut toujours éviter son utilisation en faisant un programme structuré et logique. Aucun exemple ne sera donné.

2.7 Règles d'écriture

- Choisir une présentation pour les accolades ouvrantes et utiliser tout le temps la même
  - À la ligne
  - Sur la même ligne
- Utiliser quand cela est possible un `switch case` lorsqu'il y a plus de 2 tests qui s'enchaînent
- Bien indenter
  - Automatique avec un bon éditeur
- Toujours préférer la version la plus lisible du programme
  - Exemple : éviter d'utiliser le `continue`

Questions de synthèse

1. Une condition dans une structure de contrôle doit toujours être entourée de quels symbôles ?
2. On peut transformer un `while` en `for` facilement en C, vrai ou faux ?
3. Le `for` en C permet de faire plus de choses que le *pour* en algorithmique, vrai ou faux ?
4. Que fait l'instruction `for (;;);` ?

5. Si l'on veut rendre les clauses d'un `switch case` indépendantes les unes des autres, quel mot-clé faut il ajouter à chaque fin de clause ?
6. Un `switch case` n'est pas adapté à la gestion de menus, vrai ou faux ?

Réponses : 1. De parenthèses. 2. Vrai. 3. Vrai. 4. Une boucle infinie. 5. Le mot-clé `break`. 6. Faux, c'est l'idéal !

CHAPITRE 3
LES OPÉRATEURS

3.1 Introduction

Les opérateurs sont des symboles particuliers qui permettent de faire des opérations pour créer des expressions. Ils sont différents des fonctions dans le sens où on les met directement à côté des paramètres (à gauche, à droite ou les deux), contrairement aux fonctions où l'on utilise les parenthèses. Cela permet d'écrire des expressions beaucoup plus facilement.

Si l'on voulait écrire une expression arithmétique toute simple avec des fonctions cela deviendrait vite horrible. Exemple :

```
a+b*2-x
```

s'écrirait :

```
soutraction(addition(a,multiplication(b,2),x))
```

C'est une facilité de saisie non négligeable mais gardez bien à l'esprit que le compilateur va faire cette conversion systématiquement.

3.2 Classification

En C, il existe des opérateurs unaires (à gauche ou à droite), des opérateurs binaires, et un opérateur ternaire.

Un opérateur unaire ne possède qu'un argument (qui peut donc être à droite ou à gauche). Un opérateur binaire a deux arguments, un à droite, et l'autre à gauche.

Tous les opérateurs arithmétiques standards sont binaires. Comme les fonctions, les opérateurs n'acceptent que certains types comme arguments et renvoient un type qui peut être différent.

3.3 Opérateurs arithmétiques

On peut appliquer les opérateurs standards aux entiers ainsi qu'aux flottants :

— + addition

— - soustraction
— * multiplication
— / division (entière si les deux arguments sont entiers)

Pour les entiers, on obtient le reste de la division entière avec % (qu'on appelle aussi modulo).

Il n'y a pas d'opérateur pour la puissance et la racine carrée. Il faut utiliser les fonctions de la bibliothèque `math.h` :

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

L'opérateur de soustraction - sert aussi d'opérateur « signe négatif » pour les constantes ou pour les expressions.

3.4 Opérateurs relationnels

Ils prennent des arguments entiers ou flottants et fournissent un résultat de type entier :
— 0 si la relation est fausse
— 1 si la relation est vraie

On utilise ces opérateurs pour faire des tests :
— < inférieur strictement
— > supérieur strictement
— <= inférieur ou égal
— >= supérieur ou égal
— == est égal à
— != est différent

⚠ => et =< ne sont pas corrects

3.5 Opérateurs logiques

Ils prennent comme arguments des entiers et permettent de faire des combinaisons logiques :
— && et logique
— || ou logique
— ! non logique (opérateur unaire)

3.6 Opérateurs sur les bits

Ils prennent comme arguments des entiers et permettent de faire des opérations de logique bit à bit :
— & et bit à bit
— | ou bit à bit
— ^ ou exclusif bit à bit (xor)
— << décalage de bits à gauche (multiplication par des puissances de 2)
— >> décalage de bits à droite (division par des puissances de 2)

Ce ne sont pas du tout les mêmes opérations que pour les opérateurs logiques précédents. Exemples :

Arguments		&&	&	||	|
0	0	0	0	0	0
0	1	0	0	1	1
2	3	1	2	1	3
1	2	1	0	1	3

L'utilisation d'un | à la place d'un || n'aura pas beaucoup d'impact, par contre on voit bien dans ce tableau que l'utilisation d'un & à la place d'un && peut provoquer des résultats faux en terme de valeur logique.

Exemple d'erreur classique :

```
if (valeur==1 & fin!=0)
```

La version correcte est :

```
if (valeur==1 && fin!=0)
```

3.7 Opérateur d'affectation

Le signe = n'est pas une simple instruction, il s'agit d'un opérateur binaire. À gauche se trouve une expression et à droite une expression que l'on peut affecter (*lvalue* en anglais). Typiquement, il s'agit d'une variable ou d'un élément de tableau. La valeur renvoyée par l'opérateur d'affectation est celle calculée à gauche.

Ainsi, `a=3` vaut 3 et peut être réutilisé dans une expression plus complexe :

```
b=((a=3)+1);
```

est correct et fera l'affectation de `a` à 3 puis de `b` à 4.

▶ Affectations arithmétiques

Il existe des opérateurs spéciaux condensés qui permettent de faire en même temps une opération arithmétique et une affectation, le tout sur la même variable.

Version condensée	Version normale
`a+=2`	`a=a+2`
`a-=2`	`a=a-2`
`a*=2`	`a=a*2`
`a/=2`	`a=a/2`
`a%=2`	`a=a%2`

▶ Incrémentation/décrémentation

— On appelle incrémentation le fait d'ajouter la valeur 1.
— La décrémentation est l'opération inverse qui consiste à soustraire la valeur 1.

En C, il existe des opérateurs unaires pour faire cela :
— ++ pour incrémenter
— -- pour décrémenter

Suivant que l'opérande est à gauche ou à droite, l'affectation sera faite avant ou après le calcul de l'expression.

Exemples :

```
a = 1;
b = a++; /* équivalent à b=a; a=a+1; */
printf("%d %d\n",a,b);
a = 1;
b = ++a; /* équivalent à a=a+1; b=a; */
printf("%d %d\n",a,b);
```

Affichera :

```
2 1
2 2
```

Dans tous les cas, après exécution, a aura été incrémenté. C'est donc bien l'ordre d'exécution à l'intérieur de l'expression qui est concerné. Dans le premier cas, a est incrémenté après, donc sa valeur dans l'expression est 1. Dans le deuxième, il est incrémenté avant, donc sa valeur vaut 2.

3.8 Expression conditionnelle

L'opérateur d'expression conditionnelle est ternaire, il possède donc 3 arguments :

```
(expression1?expression2:expression3)
```

Que l'on pourrait traduire en :

> « si expression1 est vraie alors retourne expression2 sinon retourne expression3 ».

Exemple, macro pour calculer le max de deux nombres :

```
#define max(x,y) (x>y?x:y)
```

3.9 Concaténation d'expressions

Il s'agit de l'opérateur virgule. Il permet de calculer de gauche à droite deux expressions, et de retourner celle de droite. On peut en grouper plusieurs :

```
a=1,b++,a+1
```

est une expression qui retourne la valeur entière 2. On utilise souvent cet opérateur dans les boucles for. Il permet de mettre plusieurs instructions comme arguments du for :

```
for (i=0,j=max-1;i<max;i++,j--)
{
   ...
}
```

3.10 Priorités des opérateurs

Pour former une expression on combine souvent beaucoup d'opérateurs, mais comment grouper ceux-ci ? En mathématique il est courant de dire que la multiplication est « prioritaire » sur l'addition :

$$a \times b + c = (a \times b) + c$$

En C, il en est de même, et une table de priorité est établie avec tous les opérateurs du langage. Il existe ainsi 15 groupes d'opérateurs. Lorsqu'il y a un doute (deux opérateurs de même priorité), on applique une associativité de gauche à droite ou de droite à gauche.

Opérateurs	Symbôles	Associativité
1 fonction / indice / champ	. () [] ->	gd →
2 opérateurs unaires	! ~ ++ -- - (type) * & sizeof	dg ←
3 mult. / div. / modulo	* / %	gd →
4 addition / soustraction	+ -	gd →
5 décalage de bits	<< >>	gd →
6 relations	< <= > >=	gd →
7 égal / différent de	== !=	gd →
8 ET binaire	&	gd →
9 OU exclusif binaire	^	gd →
10 OU inclusif binaire	|	gd →
11 ET logique	&&	gd →
12 OU logique	||	gd →
13 expression cond.	? :	dg ←
14 affectations	= += -= *= etc..	dg ←
15 concaténation d'expr.	,	gd →

▶ Exemples

Prenons l'expression arithmétique simple `a=b+c*d` et appliquons les règles une par une :
- Priorité 3 : `a=b+(c*d)`
- Priorité 4 : `a=(b+(c*d))`
- Priorité 14 : `(a=(b+(c*d)))`

Avec une expression logique simple comme `a==1 && b!=2 || !c>3` :
- Priorité 2 : `a==1&&b!=2||(!c)>3`
- Priorité 6 : `a==1&&b!=2||((!c)>3)`
- Priorité 7 : `(a==1)&&(b!=2)||((!c)>3)`
- Priorité 11 : `((a==1)&&(b!=2))||((!c)>3)`
- Priorité 12 : `(((a==1)&&(b!=2))||((!c)>3))`

⚠ On voit que l'on a appliqué d'abord la priorité au `!` et non celle du `>`. Peut-être que dans ce cas là, le programmeur voulait dire `!(c>3)`.

Exemple avec des opérateurs de même priorité. Le cas le plus fréquent est celui de l'utilisation conjointe de `/` et de `*`. Exemple : `109/10*10` :
- On commence par mettre les parenthèses à gauche puisque l'associativité est de gauche à droite : `(109/10)*10`
- Puis à droite : `((109/10)*10)`

Le résultat sera donc 100 puisqu'il s'agit de nombres entiers. Si l'on avait mis `109*10/10`, le résultat aurait été 109.

⚠ À chaque fois que vous avez un doute, ajoutez des parenthèses pour forcer l'ordre des calculs
⚠ Les erreurs les plus fréquentes
- Les opérateurs unaires sont très prioritaires
- La combinaison de && et de ||

▶ Macros et priorités

⚠ Il faut toujours mettre des parenthèses autour des définitions de macro

Exemple de macro mal construite :

```
#define max(x,y) x>y?x:y
printf("%d\n",max(2,1)+1);
```

affichera 2 et non 3 car ce sera équivalent à :

```
2>1?1:1+1
```

et l'addition est prioritaire sur l'expression conditionnelle. Si on veut blinder une macro, on fait comme ceci :

```
#define max(x,y) ((x)>(y)?(x):(y))
```

Certaines de ces parenthèses seront inutiles la plupart du temps mais on n'est jamais à l'abri d'un cas non prévu.

Questions de synthèse

1. Quelle est la différence entre | et || ?
2. L'affectation = est-t-elle un opérateur ?
3. Quelle est la différence entre `a++` et `++a` ?
4. Quand-est ce que l'on utilise le sens de l'associativité des opérateurs ?
5. Pourquoi est-t-il impératif de définir des priorités d'opérateurs ?

Réponses : 1. Le premier est un ou bit à bit alors que le deuxième est un ou logique. 2. Oui, le résultat de cet opérateur est le même que la valeur affectée. 3. `a++` évalue l'opération puis incrémente, `++a` le fait dans l'ordre inverse. 4. Quand des opérateurs ont la même priorité. 5. Sinon, on serait obligé de mettre des parenthèses partout, ce qui serait lourd.

CHAPITRE 4
LES TABLEAUX ET CHAÎNES DE CARACTÈRES

4.1 Tableaux

Un tableau est une suite ordonnée de valeurs du même type. On peut accéder à un élément du tableau grâce à une numérotation de ceux-ci. En C, un tableau de n éléments sera numéroté de 0 à n-1. Ce numéro est appelé *indice*.

▶ Déclaration

On définit un tableau grâce à sa taille et le type des élément qu'il contient :

```
type nom_tableau[taille];
```

déclare un tableau `nom_tableau` de taille `taille` contenant des éléments de type `type`. Exemple plus concret :

```
int tableau[12];
```

déclare un tableau de 12 entiers, donc dont les indices iront de 0 à 11.
Le compilateur, lorsqu'il voit cette déclaration va faire l'allocation de mémoire contiguë de 12 fois la taille d'un entier `int`.

⚠ En C ANSI, la taille du tableau doit être une constante. Ça ne peut pas par exemple être un paramètre d'une fonction ou une quelconque variable. Pour faire des tableaux de longueur variable (et surtout déterminée à l'exécution) il faut recourir à l'allocation dynamique.

▶ Utilisation

On peut manipuler les éléments du tableau un par un grâce à l'opérateur `[indice]` où `indice` est le numéro de la case :

```
int tableau[12];

tableau[0] = 1;
tableau[1] = a/2;
tableau[2] = tableau[0];
...
```

⚠ On voit dans l'exemple précédent que l'on peut utiliser un élément de tableau exactement comme une variable standard, c'est-à-dire en lecture mais aussi en écriture (c'est une *lvalue*).

▶ Utilisation dans une boucle

On utilise souvent les tableaux dans des boucles. Un compteur sert alors à adresser les éléments successifs du tableau :

```
#define N_ELEMENTS 30
...
int i;
int tableau[N_ELEMENTS];
for (i=0;i<N_ELEMENTS;i++)
{
  tableau[i] = ...;
}
```

⚠ Ne pas oublier de faire commencer cette boucle à 0 et de la terminer à n-1 (d'où le <)

▶ Initialisation

Pour initialiser un tableau (c'est-à-dire donner une valeur initiale à chacun de ses éléments) on utilise les accolades :

```
int nb_jours_dans_mois[] = {31, 28, 31, 30, 31, 30,
                            31, 31, 30, 31, 30, 31};
float taux_tva[] = {5.5, 19.6, ...};
```

Dans ce cas là, il n'y a pas besoin d'indiquer la taille du tableau, elle est automatiquement calculée par le compilateur.

4.2 Chaînes de caractères

▶ Déclaration

En C, une chaîne de caractères est un tableau de caractères (`char`). On peut donc la déclarer ainsi :

```
char chaine[100];
```

mais il faut indiquer la taille maximum de cette chaîne.
Les constantes chaînes de caractères sont construites grâce aux guillemets doubles. Exemples : `"Hello !"`, `"Ligne 1\nLigne 2\nLigne 3\n"`, `"INSA de Lyon"` sont des chaînes de caractères constantes, c'est-à-dire qu'on ne peut pas les modifier.

▶ Codage des chaînes

En C, le codage est simple, c'est un tableau de caractères (donc d'entiers d'un octet) qui se termine par un zéro '\0'. La chaîne `"Hello !"` sera donc codée par le tableau suivant :

Indice	0	1	2	3	4	5	6	7
Valeur	'H'	'e'	'l'	'l'	'o'	' '	'!'	'\0'

⚠ Il faut donc toujours n+1 caractères pour coder une chaîne de longueur n.

Un tableau peut avoir une taille plus grande que la taille réelle de la chaîne de caractères qu'il contient. En effet, la fin de la chaîne est indiquée par le '\0' qui peut donc se trouver à différentes positions dans le tableau :

- En première position, la chaîne est considérée comme vide (en constante on la note `""`)
- En dernière position, la longueur de la chaîne est maximum
- Tous les intermédiaires sont possibles

▶ Initialisation

Les constantes chaînes de caractères peuvent servir à l'initialisation d'une variable de type chaîne de caractères non constante :

```
char chaine[] = "Initialisation";
```

Dans ce cas là, la taille du tableau est automatiquement calculée et chaque caractère est recopié dans les cases du tableau.

⚠ Cette initialisation n'est possible que lors de la déclaration, si vous essayez de refaire l'opération, le compilateur ne voudra pas et affichera : « incompatible types in assignment ». Il faudra recourir à la fonction strcpy qui sera abordée plus tard.

On peut aussi faire une initialisation de chaîne en indiquant la taille maximum de la chaîne. Exemple :

```
char chaine[100] = "Initialisation";
```

Ceci crée une chaîne de caractères qui contiendra au maximum 99 caractères (on enlève un pour le '\0') mais qui est initialisée avec une chaîne de 14 caractères.

4.3 Tableaux de chaînes

Il est souvent utile de fabriquer des tableaux dont les éléments sont des chaînes de caractères. On les déclare ainsi :

```
char * intitule_mois[] = { "janvier", "février", "mars", "avril",
  "mai", "juin", "juillet", "aout", "septembre", "octobre",
  "novembre", "décembre" };
```

Ainsi, intitule_mois[0] correspondra à la chaîne de caractères "janvier".

```
for (i=0;i<12;i++)
{
  switch (intitule_mois[i][0])
  {
    case 'a':
    case 'o':
      printf("Le mois d'%s comporte %d jours\n",
        intitule_mois[i], nb_jours_dans_mois[i]);
      break;
    default:
      printf("Le mois de %s comporte %d jours\n",
        intitule_mois[i], nb_jours_dans_mois[i]);
  }
}
```

4.4 Manipulation de chaînes

▶ Saisie d'une chaîne

La fonction scanf avec comme format %s permet la saisie d'une chaîne de caractères :

```
#include <stdio.h>
char chaine[100];
scanf("%s",chaine);
```

mais elle a des inconvénients :

— La saisie s'arrête au premier espace, donc on ne peut pas saisir plus d'un mot
— il n'y a pas de moyen de vérifier les dépassements de la taille allouée.

C'est pour cela que l'on préfère utiliser la fonction `fgets` :

```
#include <stdio.h>
char chaine[100];
fgets(100,chaine,stdin);
```

Avec cette fonction, on est sûr de ne pas dépasser les 99 caractères et la saisie s'arrête à la fin du fichier (<Ctrl-D> au clavier) ou au caractère de retour à la ligne.
La bibliothèque standard `<string.h>` offre une multitude d'outils pour manipuler et analyser les chaînes de caractères.

▶ Longueur d'une chaîne

La fonction `strlen` permet de connaître la longueur d'une chaîne :

```
#include <string.h>
char chaine[] = "Ma chaine";
printf("\"%s\" a pour longueur %d\n",chaine, strlen(chaine));
```

affichera :

```
"Ma chaine" a pour longueur 9
```

⚠ Le caractère '\0' n'est pas compté par `strlen`

▶ Comparaison de chaînes

Une seule fonction permet de savoir si :
— deux chaînes sont égales, c'est-à-dire qu'elles sont identiques au caractère près
— une chaîne est inférieure à une autre au sens de l'ordre alphabétique (ou plutôt ASCII)

Voici le prototype de la fonction `strcmp` :

```
int strcmp(const char *s1, const char *s2);
```

Deux chaînes sont données en paramètres et la fonction fournit un résultat entier :
— 0 si les deux chaînes sont égales
— positif si `s1` est plus grand que `s2`
— négatif si `s1` est plus petit que `s2`

Exemples :

s1	s2	strcmp(s1,s2)
"aaaa"	"aaab"	-1
"aaaa"	"Aaaa"	32
"aaaa"	"aaaaaaa"	-97

⚠ Si on veut utiliser `strcmp` dans une comparaison de chaînes pour savoir si elles sont identiques, il faut faire une négation car la fonction renvoie 0 qui correspond au booléen faux :

```
if (!strcmp(chaine1,chaine2)) {
  /* les deux chaines sont identiques */
}
```

▶ Copie de chaînes

Il peut être utile de copier une chaîne dans une autre. C'est possible grâce à la fonction `strcpy` dont voici le prototype :

```
char *strcpy(char *dest, const char *src);
```

La chaîne `src` est copiée dans `dest` dont l'ancien contenu est perdu. La fin de chaîne (le caractère '\0') est elle aussi recopiée.

⚠ La destination est le premier paramètre. Pour vous en souvenir retenez que c'est comme une affectation.

⚠ La destination doit être un tableau de taille suffisante pour pouvoir accueillir la chaîne recopiée. Si ce n'est pas le cas, il risque d'y avoir une erreur de protection de la mémoire.

Exemple :

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

int main(void) {
    char destination[100] = "Coucou";
    printf("Avant : \"%s\"\n",destination);
    strcpy(destination, "C'est moi !");
    printf("Après : \"%s\"\n",destination);
    return 0;
}
```

affichera :

```
Avant : "Coucou"
Après : "C'est moi !"
```

4.5 Les pointeurs

Les notions de tableau et de pointeur sont très liées. En fait, il s'agit de la même chose, la seule nuance étant dans l'allocation de la mémoire.

Un pointeur est une adresse mémoire. Si l'on comparait la mémoire centrale de l'ordinateur à un énorme tableau, ce pointeur serait simplement un indice dans ce tableau.

Qu'une variable représente un pointeur ou un tableau, l'utilisation sera exactement la même, pour aller chercher le ième élément du tableau, on utilisera l'opérateur [].

```
int tab[12];
int * p = tab;
```

`tab[i]` ⇔ `p[i]`.

Nous reparlerons plus en détails des pointeurs ultérieurement.

4.6 Passages de tableaux en paramètres

Pour le passage de paramètres dans des fonctions, les tableaux sont complètement équivalents aux pointeurs. Passer un tableau en paramètre à une fonction, c'est équivalent à passer l'adresse de son premier élément. Les deux prototypes suivants sont donc équivalents :

```
int ma_fonction(int tableau[]);
int ma_fonction(int * tableau);
```

Bien que cela soit possible, on évitera de spécifier la taille du tableau quand il s'agit d'un paramètre d'une fonction. Le compilateur l'ignore tout simplement. Cela souligne qu'il s'agit d'un pointeur.

⚠ Le tableau est toujours passé par adresse, cela signifie que si son contenu est modifié par la fonction appelée, son contenu sera aussi modifié dans le contexte appelant. Il n'y a donc aucune copie des éléments du tableau (d'ailleurs, comment ferait-on puisqu'on ne connaît pas la taille ?)

⚠ En général, on indique à la fonction la taille du tableau grâce à un deuxième paramètre. Exemple :

```
void tri_tableau(int * tableau, int n_elements);
```

Pour ce qui est de l'appel de la fonction, on fera ainsi :

```
int tab[12];
...
tri_tableau(tab,12);
```

Questions de synthèse

1. Comment déclare-t-on un tableau de 3 `float` ?
2. Comment accède-t-on au premier élément de ce tableau ?
3. Comment peut-t-on l'initialiser ?
4. Une chaîne de caractères est un tableau de caractères, vrai ou faux ?
5. La chaîne `"Coucou"` nécessite combien d'octets au minimum pour être codée ?
6. Quelle bibliothèque standard du C permet de manipuler les chaînes de caractères ?
7. `strcmp(ch1,ch2)` est-t-il vrai ou faux si les deux chaînes sont identiques ?
8. Les tableaux et les pointeurs ont-t-ils un quelconque rapport ?

Réponses : 1. `float tab[3]`; 2. `tab[0]` 3. `float tab[3] = {0.0, 0.1, 0.2}`; 4. Vrai. 5. Il faut 6 caractères pour la chaîne elle-même et 1 de plus pour le `'\0'` final donc 7 caractères au total. 6. `string.h`. 7. Il vaut 0 donc est équivalent au booléen faux. 8. Oui, c'est presque la même chose (dans l'utilisation en tout cas).

CHAPITRE 5
LES ENTRÉES-SORTIES

5.1 Introduction

Entrée-sortie se traduit par input-output en anglais. C'est pourquoi la bibliothèque qui gère les entrées-sorties s'appelle `stdio.h` comme STanDard Input Output. Les fonctions contenues dans cette bibliothèque sont « évoluées » dans le sens où elles permettent de faire des opérations sur les fichiers de manière simple sans avoir trop à se soucier du fonctionnement interne.
Il existe des fonctions de plus bas niveau (dont se sert `stdio.h`) qui font des opérations beaucoup plus basiques : `unistd.h`. Nous ne les aborderons pas (en général ces fonctions portent le même nom mais sans le `f` au début : open, close, read, write, *etc.*).

5.2 Utilisation de puts

`puts` est la fonction la plus basique permettant d'afficher une chaîne de caractères sur la sortie standard.

```
char * chaine = "Bonjour !";
puts(chaine);
```

Automatiquement un caractère de fin de ligne est ajouté. Il existe aussi la fonction `fputs` qui permet d'écrire une chaîne sur n'importe quel flux de sortie. Ces deux fonctions ne permettent pas d'inclure des valeurs de variables dans la chaîne affichée.

5.3 Utilisation de printf

▶ Principe

`printf` permet d'afficher sur la sortie standard des chaînes formatées, c'est-à-dire construites à partir de variables.
Le prototype est le suivant :

```
int printf (const char *format, ... );
```

Le premier argument est la chaîne de format. Les arguments supplémentaires optionnels sont présentés suivant les directives contenues dans ce format. Rappelons qu'en C, une chaîne de caractères constante est indiquée par les guillemets doubles `"..."`. Les conversions contenues dans la chaîne de format sont indiquées par un `%`. La valeur renvoyée par `printf` correspond aux nombres de caractères imprimés.

- Exemples

```
printf ("Le numéro %d vaut %f\n", i, x);
```

affichera : Le numéro 7 vaut 60.7566 si i a pour valeur 7 et x 60.7566.

```
printf ("Somme de deux entiers : %d\n", n1 + n2);
```

affichera : Somme de deux entiers : 42 si n1 + n2 vaut 42.

```
printf ("Voici son prix : %.2f Euros\n", prix);
```

affichera : Voici son prix : 3.50 Euros si prix vaut 3.5.

- Formats

Le tableau suivant donne des exemples de formats de conversions. Pour afficher un % il suffit de doubler ce symbole (%%).

Type de donnée	Format	Exemple / observation
char	%c	'A'
unsigned char	%uc	
char *	%s	"ceci est une chaîne"
short int	%h	
unsigned short int	%uh	
int	%d	affiche en décimal
int	%o	affiche en octal
int	%x	affiche en hexadécimal
unsigned int	%ui	
long int	%ld	
unsigned long int	%uld	
float	%f, %e	réel en virgule fixe, réel avec exposant
float	%g	réel en f si possible, e sinon
double	%lf	
long double	%Lf	

- Largeur de champ

On peut spécifier entre le % et la lettre de conversion une largeur de champ. Il s'agit d'un entier. Ainsi, le nombre de caractères indiqué sera toujours affiché. Au besoin, des espaces seront ajoutés. C'est utile lorsque l'on veut que plusieurs lignes aient des colonnes alignées :

```
printf("Décimal Octal"
    " Hexadécimal\n");
for (i=0;i<12;i++)
{
  printf("%7d%6o%12x\n",i,i,i);
}
```

```
Décimal Octal Hexadécimal
      0     0           0
      1     1           1
      2     2           2
      3     3           3
      4     4           4
      5     5           5
      6     6           6
      7     7           7
      8    10           8
      9    11           9
     10    12           a
     11    13           b
```

▶ Précision

Pour les formats de flottants on peut ajouter une précision d'affichage après la virgule grâce au point. Exemple :

```
double d=0.12345;
printf("%.2f %.4f\n",d,d);
```

affichera :

```
0.12 0.1235
```

On peut combiner largeur de champ avec précision :

```
printf("Angle Cosinus\n");
for (i=0;i<=90;i+=15)
{
    printf("%5d%8.2f\n",i,cos((double)i/180.0*3.141592));
}
```

qui affichera :

```
Angle Cosinus
    0    1.00
   15    0.97
   30    0.87
   45    0.71
   60    0.50
   75    0.26
   90    0.00
```

5.4 Utilisation de scanf

scanf est le pendant de printf pour l'entrée de données formatées. Son prototype est :

```
int scanf (const char *format, ... );
```

Comme pour printf, le format est une chaîne de caractères, mais la syntaxe est différente. Le format peut contenir :

- Un blanc, une tabulation ou un retour à la ligne correspondant facultativement à un de ces caractères
- Un caractères différent de % qui doit être le même que celui lu en entrée sinon la lecture est bloquée
- Une spécification de format de conversion de la forme :
  %[*][nombre][l][caractère_de_conversion]
  décrivant la conversion du prochain champ en entrée dont le résultat sera placé à l'adresse correspondante (sauf si le * est présent, auquel cas le champ est ignoré)

Un champ en entrée est une chaîne sans blanc terminée par le premier caractère non interprétable ou de longueur spécifiée par nombre. Les caractères de conversion :

% attend un % en entrée

d, o ou x L'argument doit être un pointeur sur int

s Le champ lu se termine au premier blanc ou retour à la ligne rencontré (ils seront supprimés). La chaîne suivie du caractère nul est placée à l'adresse donnée (tableau de caractères) sans contrôle de longueur

c Dans ce cas, le champ est un caractère et les blancs ne sont pas éliminés. L'argument doit être de type pointeur sur `char`.

e ou f L'argument doit être de type pointeur sur `float`. Le champ en entrée a une forme avec ou sans décimales, avec ou sans exposant.

[liste de caractères] L'argument est un tableau de `char`. Dans ce cas le champ en entrée n'est pas limité par le blanc, mais par le premier caractère élément (non élément) de la liste si elle (ne) commence (pas) par `'^'`.

d, o et x doivent être mis en majuscule ou précédés par l si l'argument est de type `long`, de même que e et f si l'argument est de type `double`. d, o et x doivent être précédés d'un h si l'argument est de type `short`.

▶ Valeur renvoyée

EOF est renvoyé en fin de fichier (<ctrl d> sur le clavier). Le nombre de champs interprétés est renvoyé et doit être égal au nombre d'arguments passés, sinon il y a erreur.

▶ Exemples

format	entrée	retour	conversions
`" %2d:%2d:%2d"`	`"12:33:44<entrée>"`	3	12, 33, 44
`" %2d:%2d:%2d"`	`"12:33 44<entrée>"`	2	12, 33, ?
`" %2d:%2d:%2d"`	`"12:33<ctrl d>"`	2	12, 33, ?
`" %2d:%2d:%2d"`	`"<ctrl d>"`	EOF	?, ?, ?
`" %[^\n]"`	`"123 45 def<entrée>"`	1	"123 45 def"
`" %[^\n]"`	`"123 45 <ctrl d>"`	1	"123 45 "
`" %[^\n]"`	`"  123 45<entrée>"`	1	"123 45"
`" %[oui]"`	`"oui<entrée>"`	1	"oui"
`" %[oui]"`	`"ioi<entrée>"`	1	"ioi"
`" %[oui]"`	`"aiou<entrée>"`	0	?

▶ Quelques conseils

⚠ Ne pas oublier l'opérateur & devant le nom de la variable. Il permet d'obtenir l'adresse de celle-ci.

— Lorsque plusieurs `scanf` s'enchaînent, des retours chariot vont rester « coincés » dans le buffer clavier. C'est pourquoi il faut toujours mettre un blanc en début de chaîne de format (sauf cas très particulier). Cela permet d'éliminer tous les espaces, retours chariot et tabulations avant de commencer la première conversion.

— Le format `%s` permet de convertir uniquement un mot et non une ligne entière, pour avoir une ligne entière, il est préférable d'utiliser `%[^\n]` ou bien de faire de la lecture caractère par caractère.

— Il faut toujours analyser le code retour de `scanf` pour savoir si les conversions ont bien été faites, sinon on risque d'utiliser des variables qui ont une valeur aléatoire et de faire une boucle infinie.

5.5 Lecture et écriture de caractères

▶ getchar et putchar

Pour la lecture de caractères sur l'entrée standard et l'écriture sur la sortie standard, on utilise les deux fonctions suivantes :

```
int getchar(void);
int putchar(int);
```

getchar permet de lire un caractère alors que putchar en écrit un.
Pourquoi un int et non un char ou unsigned char ?

- Pour permettre de renvoyer EOF (-1), c'est-à-dire la fin de fichier (<ctrl d> au clavier)
- Pour gérer les caractères dont le code est supérieur à 128 (caractères accentués par exemple)

Pour lire un texte caractère par caractère sur l'entrée standard, on peut donc faire ce genre de boucle :

```
int c;
while ((c = getchar()) != EOF)
{
   ...
}
```

⚠ Mais attention, lire les caractère un par un ne signifie pas que l'on va savoir à quel moment l'utilisateur tape sur la touche. En effet, le buffer d'entrée n'est vidé que lorsque l'utilisateur tape sur la touche entrée.

Par défaut, un terminal affiche automatiquement ce qui est saisi par l'utilisateur (pour que celui-ci voit ce qu'il tape) mais cette fonction peut être désactivée (il faut une bibliothèque spéciale pour cela liée au terminal). Cela signifie que si vous affichez systématiquement ce qui est saisi (voir programme ci-dessous), alors l'affichage sera double :

```
while ((c=getchar()) != EOF)
{
  putchar(c);
}
```

▶ fgetc et fputc

En fait, getchar et putchar sont des macros définies à partir de fgetc et fputc dont voici les prototypes :

```
int fgetc(FILE *stream);
int fputc(int c, FILE *stream);
```

L'utilisation de flux autres que les flux stdin et stdout est expliquée dans la section suivante.

5.6 Ouverture fermeture de fichier

Pour l'instant, les opérations de lecture et d'écriture que nous avons faites concernaient uniquement l'entrée et la sortie standard stdin et stdout. Il s'agit de flux (stream en anglais) dont le type est FILE *.

Il est possible d'associer un fichier physique (c'est à dire /home/login/unfichier.txt par exemple) à un flux. C'est ce que l'on appelle l'ouverture de fichier. L'opération inverse consiste à fermer le fichier.

▶ Ouverture

Elle se fait par la fonction fopen :

```
FILE *fopen (const char *chemin, const char *mode);
```

qui prend deux arguments :

- chemin : le nom du fichier physique
- mode : le mode d'ouverture

Le mode d'ouverture est une chaîne de caractères dont voici quelques significations :

"r" lecture seule

"w" écriture seule, le fichier est écrasé s'il existait

"r+" lecture/écriture

"a" ajout de données (positionnement initial à la fin du fichier)

En plus de cela, on peut ajouter le caractère `b` pour indiquer qu'il s'agit de données binaires qui vont être écrites (norme ANSI). Sous un système POSIX comme linux, il sera ignoré.

`fopen` renvoie un descripteur de fichier ou flux, qui sera utilisable par beaucoup de fonctions de la bibliothèque `stdio.h`. Si l'ouverture a échoué, la fonction renvoie `NULL` le pointeur nul. Comment ouvrir un fichier proprement ?

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

int main(void) {
   FILE * fid;
   fid = fopen("coucou","r");
   if (!fid)
   {
      printf("Erreur : %s\n",strerror(errno));
      return 1;
   }
   fclose(fid);
   return 0;
}
```

▶ Fermeture

Lorsque l'on a terminé de lire, ou modifier un fichier, il faut impérativement le fermer car sous certains systèmes, il n'est pas automatiquement fermé à la fin de l'exécution du programme. Cela permet de finir d'écrire des données si elles avaient uniquement été stockées dans une mémoire tampon.

```
int fclose( FILE *stream);
```

La fonction renvoie un entier qui vaut 0 si tout s'est bien passé. Sinon, `EOF` est retourné et la variable `errno` est positionnée.

5.7 Lecture/écriture

Une fois qu'un fichier est ouvert, on peut le lire, écrire à l'intérieur (tout dépend de la manière dont on l'a ouvert évidemment). Il existe fondamentalement deux façons d'accéder à un fichier :

— Accès textuel : dans ce cas là, le fichier ne contient que des caractères ASCII et est affichable directement

— Accès binaire : dans ce cas là, le fichier contient des informations binaires qui sont la sauvegarde directe d'informations contenues dans les variables

Alors que l'on ne peut accéder aux fichiers textuels que de manière séquentielle, on peut accéder à un fichier binaire de manière directe en se positionnant à un endroit précis. Les fichiers binaires ont une structure qui doit être définie à l'avance, les fichiers textuels n'ont pas vraiment de structure par défaut.

▶ Accès textuel

Toutes les fonctions de lecture/écriture sur l'entrée et la sortie standard ont leur équivalent pour ce qui est des fichiers quelconques. Il suffit en général de mettre un f devant.

fprintf Il s'agit de la transposition de printf :

```
int fprintf(FILE *stream, const char *format, ...);
```

fscanf Il s'agit de la transposition de scanf :

```
int fscanf( FILE *stream, const char *format, ...);
```

fgetc Il s'agit de la transposition de getc :

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

fputc Il s'agit de la transposition de putc :

```
int fputc(int c, FILE *stream);
```

fgets Il s'agit de la transposition de gets :

```
char *fgets(char *s, int size, FILE *stream);
```

fputs Il s'agit de la transposition de puts :

```
int fputs(const char *s, FILE *stream);
```

mais le retour chariot n'est pas ajouté (contrairement à puts)

▶ Accès binaire

Il existe deux fonctions (fwrite et fread) pour écrire et lire des données binaires. On fournit à ces fonctions quatre arguments :

- Un pointeur qui va être le point mémoire qui va être lu ou écrit
- La taille d'un élément de base à copier (en octets)
- Le nombre d'éléments de base à copier
- Le flux concerné

Cela donne les deux prototypes suivants :

```
size_t fread  (void *ptr,
               size_t size,
               size_t nmemb,
               FILE *stream);
size_t fwrite (const void *ptr,
               size_t size,
               size_t nmemb,
               FILE *stream);
```

Exemple
Ecriture dans un fichier d'un tableau d'entiers.

```
#include <stdio.h>

int main(void) {
   int taille = 10;
   int tableau[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, -1};
   FILE * fid;
```

```
    fid = fopen("fichier.bin","w");
    fwrite(&taille,sizeof(taille),1,fid);
    fwrite(tableau,sizeof(int),10,fid);
    fclose(fid);
    return 0;
}
```

L'utilitaire linux `hexdump` permet de visualiser le contenu de ce fichier :

```
$ hexdump fichier.bin
0000000 000a 0000 0001 0000 0002 0000 0003 0000
0000010 0004 0000 0005 0000 0006 0000 0007 0000
0000020 0008 0000 0009 0000 ffff ffff
000002c
```

On voit que les entiers ont été codés sur 4 octets dans cet exemple et que les octets de poids faibles sont écrits en premier.

Nous verrons plus tard comment utiliser ces fonctions pour stocker des structures plus complexes en C.

Questions de synthèse

1. Quel fichier d'en-tête doit-t-on inclure pour pouvoir utiliser les fonctions standard d'entrée-sortie en C ?
2. Qu'affiche `printf("%.3f %.3f",0.1,0.0006);` ?
3. Que renvoie la fonction `scanf` ?
4. Quelle fonction permet d'ouvrir un fichier ?
5. Quelle fonction permet de fermer un fichier ?
6. Lorsque l'on crée un fichier binaire, on ne peut pas lire son contenu dans un éditeur de texte, vrai ou faux ?

Réponses : 1. stdio.h. 2. 0.100 0.001 3. Le nombre de conversions correctement effectuées. 4. fopen 5. fclose 6. Vrai.

CHAPITRE 6
FONCTIONS ET POINTEURS

6.1 Les pointeurs

Un pointeur est une adresse mémoire. Généralement, ce pointeur est contenu dans une variable et à l'endroit où il pointe se trouve une autre variable.

▶ Déclaration

Au moment de la déclaration, il faut indiquer sur quel type de données le pointeur est censé pointer. On utilise le symbole * pour déclarer un pointeur. Exemples :

```
int * p_entier;   /* un pointeur sur un entier */
char * chaine;    /* un pointeur sur un caractère */
void * pointeur; /* un pointeur sur un type inconnu */
```

▶ Obtention de l'adresse d'une variable

Pour obtenir l'adresse d'une variable, on utilise l'opérateur & placé devant celle-ci.

```
int variable;
int * pvariable;
pvariable = &variable;
```

Après cette ligne de code, la variable `pvariable` pointe sur la variable `variable`. Cet opérateur permet donc d'extraire l'adresse de la variable qu'il précède.

▶ Obtention de l'objet pointé par une adresse

C'est l'opération inverse que l'on appelle aussi opération d'indirection. Grâce à l'opérateur * on peut obtenir la variable qui se trouve à l'emplacement mémoire indiqué :

```
int variable;
int * pvariable;
pvariable = &variable;
*pvariable = 1; /* équivalent à variable = 1; */
```

▶ Fonctionnement de la mémoire

Voici l'illustration de l'utilisation de la mémoire durant l'exécution d'un petit programme simple :

```
int a;
int * pa;
a = 1234;
pa = &a;
*pa = 9999;
```

Avant même l'exécution de la première ligne de code, voici l'état de la mémoire :

Adresse	Valeur	Variable
0	?	a
1	?	pa

Cela signifie que les variables a et pa ont une place réservée en mémoire par le compilateur mais ne sont pas encore initialisées.
La première ligne va donner une valeur à a :

Adresse	Valeur	Variable
0	1234	a
1	?	pa

Tandis que la deuxième ligne va initialiser pa comme pointant vers l'adresse de a (ici 0) :

Adresse	Valeur	Variable
0	1234	a
1	0	pa

La dernière ligne modifie la valeur de a sans utiliser la variable a, elle se sert de son adresse et de l'opérateur d'indirection * :

Adresse	Valeur	Variable
0	9999	a
1	0	pa

▶ Mnémotechnique

Ce n'est pas un hasard si l'on déclare les pointeurs avec le symbole * et que ce même symbole sert d'opérateur pour obtenir l'objet pointé par une adresse. Si l'on découpe une déclaration de pointeur à un quelconque endroit, on aura de part et d'autre des choses homogènes :

```
int* | pointeur;
int  | *pointeur;
```

pointeur est de type int* soit un pointeur sur entier et *pointeur est bien de type entier.

▶ Opérations arithmétiques sur les pointeurs

D'un point de vue technique, un pointeur s'apparente fortement à un entier (un indice dans le tableau énorme de la mémoire centrale). On peut donc faire tout un tas d'opérations arithmétiques sur celui-ci : additions, soustractions, incrémentations, *etc.* Cependant, il faut faire très attention à ce que cela veut dire. En effet, une incrémentation sur un pointeur ne signifie pas « ajouter un » mais « obtenir le prochain élément du type sur lequel on pointe ». Le résultat ne sera donc pas le même suivant le type sur lequel le pointeur pointe.

```
int * pentier = NULL;
double * pdouble = NULL;
printf("%x %x\n",pentier+1,pdouble+1);
```

affichera :

```
0x4
0x8
```

Ce qui correspond bien à ajouter `sizeof(type)` où `type` est le type pointé.
On peut faire toutes les opérations arithmétiques d'addition et de soustraction sur un pointeur et un entier. Par contre, on s'interdit d'additionner deux pointeurs. On peut donc manipuler des tableaux sans jamais avoir recours à l'opérateur [] :

```
int tab[5] = {1, 3, 5, 7, 9};
int * p, i;
p = tab;

for (i=0;i<5;i++)
{
    printf("%d ",*(p++));
}
```

qui affichera :

```
1 3 5 7 9
```

▶ Pointeurs et tableaux

C'est pratiquement la même chose dans l'utilisation, c'est seulement de manière interne que le compilateur fait la différence. Ainsi, on peut utiliser de la même façon les opérateurs des tableaux pour les pointeurs :

```
int * p;
int tab[10];
p = tab;
```

Les notations suivantes sont complètement équivalentes :

`tab[0]`	`tab[2]`	`tab+5`
`*tab`	`*(tab+2)`	`&(tab[5])`
`p[0]`	`p[2]`	`p+5`
`*p`	`*(p+2)`	`&(p[5])`

La seule chose que l'on s'interdit pour des tableaux et que l'on autorise pour des pointeurs, c'est de modifier la valeur :

Autorisé	Interdit
`p = p + 1;`	`tab = tab + 1;`

6.2 Passages de paramètres

▶ En C : passage par valeur tout le temps

En C, tous les passages de paramètres de fonctions se font par valeur, c'est à dire qu'une copie de l'argument est faite puis mise dans la pile. Les modifications faites par la fonction sur les arguments ne sont donc pas répercutées dans le contexte appelant. En C, le passage d'argument par adresse doit être explicite, c'est à dire que l'on doit réellement passer l'adresse de la variable que l'on veut modifier.

▶ Passage de l'adresse

Passage par valeur

```
void passage_valeur(int arg)
{
  arg= 1;
}
...
int i = 0;
passage_valeur(i);
printf("%d\n",i);
```

Affichera : 0

Passage de l'adresse

```
void passage_adresse(int *arg)
{
  *arg= 1;
}
...
int i = 0;
passage_adresse(&i);
printf("%d\n",i);
```

Affichera : 1

▶ Quand utiliser l'adresse ?

Il faut utiliser le passage de l'adresse dans les cas suivants :

- On veut modifier la valeur de l'argument et disposer de la nouvelle valeur dans le contexte appelant
- L'argument est une grosse structure qui demanderait une grande place dans la pile des arguments (copie entière de la structure). Dans ce cas là on passe souvent l'adresse de la structure en paramètre. On prend soin d'ajouter le mot-clé `const` pour préciser que l'on ne veut pas modifier le contenu le cas échéant. Exemple :

  ```
  void fonction(const STRUCT * param);
  ...
  STRUCT s;
  fonction(&s);
  ```

  Si l'on modifie le contenu de la structure, l'avertissement suivant sera produit : « warning : assignment of read-only member 'a' »

▶ Le cas des tableaux

Les tableaux sont déjà des pointeurs, donc le passage d'un tableau en paramètre est le passage de l'adresse de son premier élément.

On ne peut pas passer par valeur un tableau à moins de le copier entièrement soi-même. Ceci est dû à la nature même de la construction des tableaux.

▶ Passage *de* l'adresse

On parlera en C de passage *de* l'adresse plutôt que de passage par adresse. En effet, pour faire l'équivalent du passage par adresse vu en algorithmique, il faut faire un passage de l'adresse par valeur. C'est une nuance qu'il est très bon de garder à l'esprit.

6.3 Variable et portée

Nous avons déjà abordé la déclaration d'une variable. Il existe différent types de déclarations de variables, suivant l'endroit où elle intervient mais aussi en fonction de mots-clé que l'on peut insérer avant celle-ci. La portée d'une variable indique la zone dans laquelle celle-ci est définie. On peut dire par exemple que la portée d'une variable locale est limitée à la fonction dans laquelle elle est déclarée.

▶ Variables locales

Ce sont les plus courantes. Elle sont déclarées dans une fonction (au début), sont automatiquement allouées au début de la fonction et sont détruites automatiquement à la fin de l'exécution de la fonction. C'est le type de variable qu'il faut privilégier pour une programmation propre. Deux variables locales peuvent très bien avoir le même nom dans deux fonctions différentes. Ce sera bien deux variables différentes.

▶ Variables globales

La déclaration d'une variable globale intervient en dehors de toute fonction. C'est une variable qui sera définie ensuite dans tout le source et accessible à toutes les fonctions. Si une déclaration de variable locale intervient, elle aura la priorité sur une variable globale :

```
#include <stdio.h>

int i=0; /* à ne pas faire ! */

int main (void){
    int i=1;
    printf("Valeur de i : %d\n",i);
    return 0;
}
```

Affichera : `Valeur de i : 1`.
Les variable globales sont à proscrire dans toute programmation structurée. En effet, elles masquent le passage de paramètres qu'il y aurait dû avoir (ou le retour d'une fonction) et rendent moins lisible le code.

▶ Variables externes

Ce sont des variables globales partagées par plusieurs fichiers sources. On indique par le mot-clé `extern`, qu'il ne faut pas allouer de mémoire pour cette variable car elle est définie dans un autre fichier. La résolution sera donc faite au moment de l'édition des liens. Encore pire que les variables globales, on ne devrait jamais utiliser ce genre de variables. Pourtant, les mécanismes d'erreurs (la variable `errno` par exemple) l'utilisent pour supprimer un paramètre à tous les appels de fonction.
Exemple :

```
extern int max;
```

▶ Variables statiques

Une variable statique est une variable locale qui est allouée en début de programme et libérée en fin de programme. Cela signifie qu'une variable statique gardera sa dernière valeur d'un appel au suivant. Exemple :

```
#include <stdio.h>
int fonction_etrange(void) {
    static int val = 0;
    val ++;
    return val;
}
int main (void)
{
    int val1,val2;
    val1 = fonction_etrange();
    val2 = fonction_etrange();
    printf("%d, %d\n",val1,val2);
    return 0;
}
```

Affichera : 1, 2.

⚠ Cela signifie que l'appel d'une fonction dépend d'autre chose que de ses arguments. C'est un comportement qu'il faut à tout prix éviter pour de la programmation structurée.

▶ En bref

— Privilégier les variables locales
— Ne pas utiliser les variable globales, statiques, ou externes, sauf en cas d'extrême nécessité (par exemple, utilisation d'une bibliothèque qui y oblige)

6.4 Les macros

La commande du préprocesseur `#define` permet de définir des constantes symboliques. Elle permet aussi lorsqu'on ajoute des arguments de définir des macros. C'est le préprocesseur qui va effectuer le travail, donc il ne s'agit que de transformation de code. Exemples classiques :

```
#define min(x,y) ((x<y)?(x):(y))
#define abs(x) ((x>=0)?(x):(-(x)))
```

Le code suivant :

```
resultat = min(6,abs(-2));
```

sera transformé ainsi :

```
resultat=((6<((-2>=0)?(-2):(-(-2))))?(6):(((-2>=0)?(-2):(-(-2)))));
```

▶ Pourquoi tant de parenthèses ?

Le code généré contient beaucoup de parenthèses, c'est parce que la définition de départ en contenait aussi beaucoup.

⚠ Ces parenthèses sont indispensables : elles permettent d'éviter les effets de bords inhérents aux macros.

Prenons un exemples très simple de macro mal définie :

```
#define abs(x) x>0?x:-x
```

Si l'on essaye de l'appliquer à l'expression suivante :

```
resultat = abs(2-6);
```

Cela donnera :

```
resultat=2-6>0?2-6:-2-6;
```

Ce qui donnera comme résultat -8 (un peu gênant pour une valeur absolue). On voit très facilement qu'avec des parenthèses, le problème n'existe pas.

▶ Traces automatiques de débugage

Le préprocesseur définit des variables spéciales qui correspondent :
- Au nom du fichier traité : `__FILE__`
- Au numéro de ligne courant : `__LINE__`

On peut donc imaginer une macro qui affiche ces informations en plus d'un message pour indiquer où l'on se trouve dans le programme au moment de l'exécution :

```
#define trace(affichage) printf("%s:%d -> %s\n", \
            __FILE__,__LINE__,__STRING(affichage))
```

Ainsi `trace(controle ok);` affichera quelque chose dans le genre :

```
fichier.c:14 -> controle ok
```

▶ Traces plus élaborées

Pour faire des traces plus élaborées, on peut utiliser les clauses `#ifdef` du préprocesseur et modifier la définition de la macro `trace` selon qu'une constante est définie ou non :

```
/* fichier debug.h */
#ifndef printf
# include <stdio.h>
#endif

#ifdef DEBUG
# define trace(expr) printf("%s:%d -> %s\n", \
              __FILE__,__LINE__,__STRING(expr))
#else
# define trace(expr) ((void) 0)
#endif
```

Cela signifie que l'on doit définir `DEBUG` avant d'inclure le fichier `debug.h`. Dans le cas contraire, la définition de `trace` se limitera à ne rien faire (grâce à `((void) 0)`). Les trois premières lignes servent à inclure le fichier `stdio.h` si la commande `printf` est inconnue.

> Note : la commande `__STRING` du préprocesseur permet de transformer n'importe quel argument (le préprocesseur n'a aucune notion de types de données) en chaîne de caractères constante en ajoutant les guillemets doubles `"` autour.

Questions de synthèse

1. Le résultat de l'addition d'un entier à un pointeur ne dépend pas du type du pointeur, vrai ou faux ?

2. Peut-t-on programmer en C sans jamais utiliser les [] sauf pour les déclarations de tableaux ?
3. Existe-t-il vraiment un passage par adresse en C ?
4. Quelle est la portée de variable qu'il faut privilégier en C ?
5. À quoi faut-t-il faire attention lorsque l'on programme une macro ?

Réponses : 1. Faux, suivant la taille du type, le résultat sera différent. 2. Oui, car tab[i] est équivalent à *(tab+i). 3. Non, tous les passages d'arguments se font par valeur. 4. La portée locale. 5. À bien mettre des parenthèses pour éviter les effets de bord.

CHAPITRE 7
INTERFACE AVEC LE SYSTÈME

7.1 Arguments de la ligne de commande

L'interaction avec la ligne de commande se fait grâce aux arguments que reçoit la fonction `main`. Voici son prototype :

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

▶ Nombre d'arguments

Le premier argument `argn` indique le nombre d'arguments qu'a reçu le programme lors de l'appel en ligne de commande. Ce nombre d'arguments comprend le nom du programme lui-même.
Exemple :

```
$ programme argument1 argument2
```

donnera 3 comme valeur de `argn`.

▶ Tableau de chaîne des arguments

Chacun des arguments est ensuite stocké sous forme d'un chaîne de caractères. Toutes les chaînes de caractères sont ensuite regroupées dans un tableau qui est le deuxième paramètre `argv` de la fonction `main`. Ainsi, dans l'exemple précédent :

- `argv[0]` vaut `"programme"`
- `argv[1]` vaut `"argument1"`
- `argv[2]` vaut `"argument2"`

Les arguments de la ligne de commande sont séparés par des espaces. Si vous désirez inclure un espace dans un des arguments, vous pouvez faire de deux manières :

- Le protéger par un anti-slash
- Inclure l'argument entier entre guillemets doubles

Le programme suivant permet d'afficher ses arguments :

```
#include <stdio.h>
int main(int argn,char * argv[]) {
    int i;
    for (i=0;i<argn;i++) {
```

```
        printf("argv[%d] = %s\n",i,argv[i]);
    }
    return 0;
}
```

Quelques exemples d'exécution :

```
$ arg argument1 argument2
argv[0] = arg
argv[1] = argument1
argv[2] = argument2
$ arg argument1\ argument2
argv[0] = arg
argv[1] = argument1 argument2
$ arg "argument1 argument2"
argv[0] = arg
argv[1] = argument1 argument2
```

▶ Bibliothèque de traitement

Il est courant d'utiliser un mécanisme d'options de ligne de commande. Ces options sont représentées par un caractère et peuvent induire un paramètre supplémentaire. Elles peuvent être obligatoires ou optionnelles.

Le C ANSI ne spécifie pas de commande pour analyser automatiquement les options que l'on peut mettre en ligne de commande. Pour cela, il faut avoir recours à des fonctions comme `getopt` définies par le standard POSIX.
La fonction `getopt` prend comme argument `argn` et `argv` (les deux paramètres du `main`) ainsi qu'une chaîne qui indique le format attendu. En retour, elle renvoie un caractère d'option, ':' pour indiquer qu'il manque un paramètre, '?' pour une option inconnue ou -1 si toutes les options ont été analysées. En plus de cela, elle utilise les variables globales externes `optarg`, `optind`, `opterr` et `optopt`. Reportez vous à la documentation pour plus de détails.

7.2 Code retour système

La fonction `main` renvoie un entier qui représente le code retour système qui sera communiqué au processus qui a déclenché l'exécution du programme. Ce code doit être :

- Nul si le programme s'est bien terminé
- Différent de 0 dans un autre cas (par exemple exécution arrêtée car le nombre d'arguments en ligne de commande est insuffisant)

La bibliothèque standard `stdlib.h` définit les constantes suivantes :

- `EXIT_SUCCESS` qui vaut 0
- `EXIT_FAILURE` qui vaut 1

Il est préférable d'utiliser ces constantes plutôt que directement les valeurs 0 et 1. Cela rend le code plus lisible.

7.3 Exemple

Voici un exemple de programme typique prenant des arguments en ligne de commande :

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

int main(int argn,char * argv[]) {
    int arg;
    if (argn<2) {
        printf("Usage : %s argument\n",argv[0]);
        printf("Où argument est un entier\n");
        return EXIT_FAILURE;
    }
    arg = atoi(argv[1]);
    /*...*/
    return EXIT_SUCCESS;
}
```

7.4 Utilisation du code retour

Le code retour peut être récupéré par les fonctions de la famille de wait qui interrompent l'exécution d'un programme en attendant la fin de l'exécution d'un autre processus. Le code retour sera alors communiqué et pourra être utilisé.
Une autre façon d'utiliser le code retour est dans la construction de script shell. Par exemple, un programme qui ne s'est pas bien terminé sera considéré comme faux. Dans l'exemple précédent :

```
$ line && echo "Ok"
Usage : line argument
Où argument est un entier
$ line 12 && echo "Ok"
Ok
```

7.5 La fonction exit

La fonction exit permet d'arrêter à tout moment (même sans se trouver dans le main) l'exécution du programme et de donner un code retour au système :

```c
exit(EXIT_FAILURE);
```

est équivalent à :

```c
return EXIT_FAILURE;
```

dans le main.

Questions de synthèse

1. Peut-t-on changer le prototype de la fonction main ?
2. Le premier argument de la fonction main peut être inférieur à 1, vrai ou faux ?
3. Si un programme est appelé ainsi : programme arg1 arg2, quels seront les arguments de la fonction main ?
4. La fonction main renvoir un entier, à quoi correspond-t-il ?
5. exit et return sont équivalents, vrai ou faux ?

Réponses : 1. Non, surtout pas. On peut par contre ne pas déclarer ses paramètre si on ne s'en sert pas. 2. Non, car au minimum, il y a forcément le nom du programme éxécutable. 3. `argn` vaut 3, `argv[0]` vaut `"programme"`, `argv[1]` vaut `"arg1"`, et `argv[2]` vaut `"arg2"`. 4. C'est le code retour renvoyé au système à la fin de l'éxécution. 5. Ils sont équivalents seulement lorsqu'ils sont utilisés dans le `main`. De plus l'un est une fonction alors que l'autre est un mot-clé.

CHAPITRE 8
TYPES ÉVOLUÉS

8.1 Définition d'un nouveau type

▶ La commande typedef

Il est possible de définir des raccourcis vers de nouveaux types avec la commande `typedef`. La syntaxe :

```
typedef <un type existant> <nouveau type>;
```

Exemple : le type booléen pourrait être défini ainsi

```
typedef int booleen;
```

⚠ En réalité, la bibliothèque `<stdbool.h>` le définit grâce à une commande du préprocesseur `#define` car dans ce cas là, cela ne pose aucun problème

Ainsi, on pourrait déclarer une variable booléenne grâce à :

```
booleen variable_booleenne;
```

Ou même un tableau de booléens :

```
booleen tableau_de_booleens[20];
```

Voici d'autres exemples :

```
/* Entier voudra dire int */
typedef int Entier;

/* type Adresse = pointeur vers char */
typedef char *Adresse;

/* Hauteur est complètement équivalent à int */
typedef int Hauteur;
```

Pourquoi ne pas utiliser `int` directement ?

— Pour plus de lisibilité, on attache une sémantique (un sens) au type de données
— Pour plus d'évolutivité. On peut s'apercevoir que finalement, il vaut mieux coder la hauteur sur un flottant, il suffit alors de changer une seule ligne de code.

▶ Des types qui représentent des tableaux

On peut utiliser `typedef` pour faire des raccourcis vers des types qui représentent des tableaux. Attention à la syntaxe :

```
typedef char Chaine20[21];
```

Dans ce cas là, `Chaine20` est un nouveau type qui représente un tableau de 21 caractères. On peut donc déclarer des variables comme ceci :

```
Chaine20 nom, prenom;
```

▶ typedef vs. #define

Tout d'abord, attention, la syntaxe est inversée :

```
#define Hauteur int
```

est équivalent à :

```
typedef int Hauteur;
```

⚠ Ne pas oublier le ; à la fin d'une `typedef`
⚠ Ne surtout pas mettre de ; à la fin d'un `#define`, cela provoque des erreurs très dures à détecter

Les principales différences :

— `#define` est une directive du préprocesseur, cela signifie que lors de la compilation, toutes les occurrences du nouveau type auront déjà été remplacée par le type équivalent
— `typedef` est une instruction du C
— `#define` ne permet pas de définir des types de tableaux
— `#define` est normalement plus rapide
— `typedef` est surtout utilisé pour définir des types qui sont des raccourcis vers des structures (voir la suite du cours)

Par convention, on notera tout nouveau type avec une majuscule au début.

8.2 Les structures

▶ Syntaxe

```
struct nom_structure
{
    type membre1;
    type membre2;
    ...
};
```

On l'utilise comme ceci :

```
struct nom_structure a, b;
```

▶ Exemple

Pour définir une structure de date :

```
struct date
{
    int jour;
    int mois;
    int annee;
};
```

Que l'on utilise comme ceci :

```
struct date d1, d2;
```

▶ Type associé à une structure

On peut construire un nouveau type associé à une structure grâce à la commande `typedef` vue précédemment :

```
typedef struct
{
    int jour;
    int mois;
    int annee;
} Date;
```

⚠ On remarquera que dans ce cas-là, on n'a pas besoin de mettre explicitement un nom à la structure
Une fois que le type est défini, la déclaration de variable est simplifiée :

```
Date d1, d2;
```

et le mot-clé `struct` n'a plus besoin d'être présent. On peut imbriquer des structures les unes dans les autres :

```
typedef struct
{
    char nom[30];
    char adresse [60];
    Date date_naiss;
    Date date_ouv_cpte;
} Client;
```

▶ Accès aux membres

L'intérêt est d'avoir une seule variable pour décrire plusieurs caractéristiques. Pour accéder à chacune de ces caractéristiques, on utilise l'opérateur . :

```
Client c1, c2;

strcpy(c1.nom , "DUPONT" ); /* pour les chaînes déjà allouées
                               la copie est obligatoire */
strcpy(c1.adresse , "7 rue Imbert Colomes 69001 LYON");
c1.date_naiss.jour = 14;  /* affectation membre après membre */
c1.date_naiss.mois = 01;
c1.date_naiss.annee = 1975;
c2.date_naiss = {25,9,78};  /* affectation en bloc */
```

▶ Pointeurs sur structure

Étant donné que les structures sont des variables, elles sont implantées en mémoire et possèdent donc une adresse. L'opérateur & permet d'obtenir cette adresse :

```
Date *pdate, date_comm;
pdate = &date_comm;
```

Pour accéder aux membres de la structure directement à partir du pointeur, on utilise l'opérateur -> à la place du point :

```
pdate->jour += 15;
```

qui est absolument équivalent à :

```
(*pdate).jour += 15;
```

⚠ Les opérateurs . et -> sont très prioritaires, donc il faut systématiquement mettre des parenthèses

```
++ pdate->jour;
```

est interprété comme :

```
++ (pdate->jour);
```

et non comme :

```
(++pdate)->jour;
```

Dans le premier cas, on incrémente le membre `jour` pointé par `pdate`, dans le second, on incrémente `pdate` et on va chercher le membre `jour` associé.

▶ Taille d'une structure

Il est utile de savoir quelle taille prend une structure en mémoire. On utilise `sizeof` pour ça :

```
sizeof (Date)
```

ou

```
sizeof *pdate
```

On peut donc utiliser indifféremment le nom d'un type ou le nom d'une variable.

⚠ La taille d'une structure n'est pas forcément égale à la taille des membres qu'elle contient. Il y a en effet des contraintes d'alignement.

Exemple :

```
typedef struct {
   char membre1;
   int membre2;
} Structure;

printf("Structure : %d\n",sizeof(Structure));
/* affiche 8 */
printf("Somme des membres : %d\n",sizeof(char)+sizeof(int));
/* affiche 5 */
```

Ce résultat a été obtenu sous linux x86 avec gcc mais peut varier en fonction de l'architecture de la machine et du compilateur. Ici, on voit clairement que l'alignement s'est fait sur 4 octets.

▶ Tableaux dans des structures

On peut mettre des tableaux dans des structures :

```
struct etablissement
{
    char nom[30];
    float reductions [3];
    int ref;
};
```

Mais il faut faire attention à l'initialisation. On peut considérer le membre comme une variable normale, donc :

```
struct etablissement insa, lyon1;
insa.nom = "INSA de Lyon";
```

n'est pas possible et ne pourra pas être compilé. En effet, `"INSA de Lyon"` est un pointeur vers une zone mémoire contenant la chaîne, alors que `insa.nom` est de type tableau. Par contre on pourra faire :

```
struct etablissement insa, lyon1;
strcpy(insa.nom,"INSA de Lyon"); /* initialisation par copie */
insa.reductions [0] = 5.0;
insa.reductions [1] = 18.6;
insa.reductions [2] = 20.0;
insa.ref = 1820001;

lyon1.nom[0] = 'U'; /* initialisation élément par élément */
lyon1.nom[1] = 'C';
lyon1.nom[2] = 'B';
lyon1.nom[3] = 'L';
lyon1.nom[4] = '\0';
lyon1.reductions [0] = ...
```

▶ Fonction retournant une structure

Le type de retour d'une fonction peut être une structure. C'est un moyen simple de retourner plusieurs valeurs en même temps. Exemple :

```
Date aujourdhui(void);
```

est une fonction qui renvoie un paramètre de type `Date` et qui ne prend aucun paramètre. Souvent, on utilise des fonctions qui renvoient un pointeur vers une structure :

```
Date * paujourdhui(void);
```

Cela nécessite de faire de l'allocation dynamique mais est très pratique. En outre, on peut renvoyer la valeur `NULL` en cas d'échec.

▶ Tableaux de structures

Il peut être utile de considérer plusieurs structures. Une façon de les organiser est de les mettre dans un tableau :

```
struct motcle
{
  char mot[20];
  int cpt;
};
struct motcle tabcle [NCLES];
```

où `NCLES` est une constante représentant le nombre d'éléments du tableau. On peut le définir ainsi par exemple :

```
#define NCLES 35
```

On peut initialiser un tableau facilement de la manière suivante :

```
struct motcle tabcle[] = {
  "break", 0,
  "case",0,
  "char", 0,
```

```
  "continue",0,
  "if", 0,
  "for",0,
  "return", 0,
  "void",0,
  "while", 0,
};
```

Le nombre d'éléments sera automatiquement calculé par le compilateur. On peut alors déterminer ce nombre d'éléments grâce à la macro suivante :

```
#define NCLES (sizeof tabcle / sizeof (struct motcle))
```

ou encore mieux :

```
#define NCLES (sizeof tabcle / sizezof (tabcle[0]))
```

On accède aux membres des éléments du tableau de la façon suivante :

```
if (strcmp (tabcle[i].mot, mot_lu) == 0)
  tabcle[i].cpt ++;
```

▶ Opérations sur les structures

Les seules opérations autorisées sur les structures sont :

- la copie
- l'affectation en la considérant comme un tout
- la récupération de son adresse avec l'opérateur &
- l'accès à ses membres grâce à .

On peut les utiliser comme argument de fonctions, ou comme retour. On peut initialiser une structure lors de sa déclaration ou lors d'une affectation.

⚠ On ne peut pas comparer deux structures. Il faudra donc implémenter des fonctions spéciales pour ça.

Exemple :

```
typedef struct {
  char nom[20];
  int hauteur;
  int largeur;
  int longueur;
} Colis;

int egalite_colis_taille(Colis * c1, Colis * c2) {
  return ( (c1->hauteur == c2->hauteur) &&
           (c1->largeur == c2->largeur) &&
           (c1->longueur == c2->longueur) );
}
```

Ici, on ne compare pas tous les membres de la structure, car le nom nous importe peu. Mais, ça pourrait être l'inverse.

▶ Fichiers séquentiels

Un fichier séquentiel est un fichier binaire dans lequel on range directement des structures de données du même type. On peut ainsi accéder directement à un des enregistrement puisque chacun d'entre eux fait la même taille.

Exemple de lecture :

```
typedef struct {
    ...
} Client;
FILE * fid;
Client c;
fid = fopen("fichier_clients.bin","rb");
/* on veut accéder au quatrième enregistrement */
position = 3;
fseek(fid,sizeof(Client)*position,SEEK_SET);
fread(&c,sizeof(Client),1,fid);
```

Exemple d'écriture :

```
strcpy(c.nom,"Dupont");
...
fid = fopen("fichier_clients.bin","wb");
/* on veut écrire sur le premier enregistrement */
position = 0;
fseek(fid,sizeof(Client)*position,SEEK_SET);
fwrite(&c,sizeof(Client),1,fid);
```

On voit bien sur cet exemple que la lecture et l'écriture se font exactement de la même manière.

8.3 Les énumérations

▶ Définition

Une énumération est une variable définie par une liste de valeurs entières constantes, auxquelles on attribue des noms symboliques. Elle est définie grâce au mot-clé enum. On peut l'utiliser comme struct, c'est à dire avec ou sans typedef.

```
enum mon_enumeration {NOM1, NOM2, NOM3, ... };
```

définit une énumération, tandis que :

```
typedef enum {NOM1, NOM2, NOM3, ...} Type_enumeration;
```

définit un type raccourcis vers une énumération. Exemple :

```
typedef enum {NON, OUI} Logique;
```

Dans ce cas là, NON vaudra 0 et OUI vaudra 1.

▶ Utilisation

On peut ensuite utiliser ce nouveau type :

```
Logique rep;
if (rep == NON).
   ...
```

Les mnémoniques qui définissent le type énuméré prennent des valeurs entières commençant à 0 par défaut et incrémentées automatiquement. Pour commencer à une autre valeur :

```
enum mois {JAN=1, FEV, MAR, AVR, MAI, JUIN,
           JUIL, AOU, SEPT, OCT, NOV, DEC};
```

Ainsi, JAN vaut 1, FEV vaut 2, etc. Les variables énumérées s'utilisent exactement comme des variables de type entier :

```
enum mois mm, *pm;

mm = NOV;
if (mm == NOV)     /* ou mm == 11 */
     printf("Novembre ! \n");
     pm = &mm;
     *pm = DEC;
     printf(" *pm = %d \n", *pm);
```

▶ Application

On peut indiquer une valeur explicitement pour chaque mnémonique :

```
typedef enum Erreurs
{
     PasErreur      = 0,
     ErreurVersion    = 101,
     ErreurMemoire    = 102,
     ErreurEcriture     = 103,
     ErreurLimite     = 110,
     ErreurNbParam    = 111,
}TYPERR;
```

C'est très pratique pour faire de la gestion d'erreur.

▶ Affichage de la valeur d'un mnémonique

⚠ On ne peut pas afficher le nom d'un mnémonique par l'intermédiaire de `printf` directement.

```
enum {FAUX, VRAI} booleen = FAUX;
printf("%d\n",booleen);
```

affiche 0 car pour lui c'est un entier. Pour pouvoir afficher le nom d'un mnémonique, il faut définir explicitement un tableau contenant les noms des mnémoniques :

```
enum {FAUX, VRAI} booleen = FAUX;
char * booleen_label[] = {"FAUX", "VRAI"};
printf("%s\n",booleen_label[booleen]);
```

affiche `FAUX`. Il existe un moyen d'automatiser la création du tableau ainsi que celle du type énuméré grâce au préprocesseur (cf. Annexe du poly, FAQ C question 6.10).

8.4 Les unions

Une union peut prendre à tout moment dans le programme des valeurs de différents types. On fait correspondre à chaque type un nom de membre. Les opérations que l'on peut faire sont les mêmes que sur une structure. La seule différence, c'est qu'un seul membre peut servir à la fois.

▶ Déclaration

```
union {
  type1 nom_membre1;
  type2 nom_membre2;
  ...
  typen nom_membren;
} variable;
```

Exemple concret :

```
union {
  int entier;
  float flottant;
  char caractere;
} variable;
```

▶ Utilisation

On peut ensuite utiliser n'importe quel membre en lecture et en écriture, mais attention, si vous écrivez par l'intermédiaire d'un membre et lisez grâce à un autre membre, vous risquez d'avoir des surprises :

```
union {
   int entier;
   float flottant;
   char caractere;
} v;
v.entier = 1;
printf("%d %f\n",v.entier, v.flottant);
v.flottant = 1.0;
printf("%d %f\n",v.entier, v.flottant);
```

affichera :

```
1 0.000000
1065353216 1.000000
```

car le codage des entiers n'est pas du tout le même que celui des flottants.

⚠ Vous devez donc savoir à tout moment quel est le type qui est contenu dans l'union pour pouvoir vous en servir.

▶ Utilisation avec typedef

Bien évidemment, on peut créer un nouveau type correspondant à une union grâce à typedef :

```
typedef union {
  type1 nom_membre1;
  type2 nom_membre2;
  ...
  typen nom_membren;
} nom_type;
```

et ensuite déclarer directement une variable avec :

```
nom_type variable;
```

▶ Taille d'une union

La taille d'une union est le maximum de la taille des membres qui la compose. Il se peut que pour certains des membres, tout cet espace ne soit pas utilisé.

8.5 Structures auto-référentielles

Certaines constructions nécessitent qu'un des membres d'une structure soit une référence à lui-même, c'est le cas pour les listes chaînées, les arbres, les graphes, *etc.* En C, on ne peut pas le faire directement

avec un typedef, il faut pour cela ajouter ce que l'on appelle une étiquette (ou *tag* en anglais) de la façon suivante :

```
typedef struct tag_nomstructure {
  struct tag_nomstructure * ref;
  ...
} nomstructure;
```

En effet, la structure en tant que nouveau type n'existe pas encore lorsque l'on décrit son contenu. Par contre, le fait de rajouter une étiquette permet de construire un type que l'on peut directement adresser par `struct tag_nomstructure`. On rajoute le pointeur car cela est beaucoup plus commode à gérer.

▶ Exemple : liste chaînée

Une liste simplement chaînée sera déclarée ainsi :

```
typedef struct tag_liste {
  struct tag_liste * suivant;
  ... /* autres membres */
} LISTE;

LISTE * liste;
```

▶ Exemple : arbre binaire

Un arbre binaire sera codé ainsi :

```
typedef struct tab_arbre_bin {
  struct tag_arbre_bin * branches[2];
  ... /* autres membres */
} ARBRE_BINAIRE;

ARBRE_BINAIRE racine;
```

8.6 Tableaux multi-dimensionnels

On connaît maintenant bien les tableaux à une dimension. En C, on peut construire des tableaux à plusieurs dimensions. Un tableau à deux dimensions correspond à un tableau avec des lignes et des colonnes. À trois dimensions, on peut voir cela comme des tableaux à deux dimensions empilés les uns sur les autres. Au delà de trois, cela devient plus difficile à imaginer et vous aurez rarement à en manipuler.

▶ Déclaration

Pour les déclarer, il suffit de mettre plusieurs fois des crochets. Exemple :

```
float matrice[3][4];
```

déclare un tableau à deux dimensions dont les indices vont de 0 à 2 tous les deux.

⚠ Il ne faut pas confondre la dimension d'un tableau (ici 2) avec sa taille (3 fois 4)

Initialisation

Il y a deux manières d'initialiser un tableau multi-dimensionnel :

```
float matrice[3][4] = {{0,1,2,3}, {4,5,6,7}, {8,9,10,11}};
float matrice[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

Elles sont complètement équivalentes.

Utilisation

On l'utilise exactement comme un tableau normal, il suffit de mettre plusieurs fois les crochets avec les indices correspondants :

```
matrice[0][1] = 1.0;
matrice[2][2] = matrice[1][1] + 3.0;
```

⚠ La notation `matrice[0,0]` est interdite.

Passage en paramètre de fonctions

Lors de la déclaration ainsi que lors de la définition d'une fonction, les paramètres qui sont des tableaux multidimensionnels doivent faire apparaître les tailles pour chacune des dimensions sauf pour la première. Exemple :

```
float calcul(float matrice[][4]);
```

C'est indispensable au compilateur pour connaître la taille de chacun des éléments qui composent le macro-tableau à une dimension. Par contre, le nombre d'éléments de ce macro-tableau n'a pas besoin d'être connu. Ceci est lié au codage des tableaux multi-dimensionnels.

Exemple classique : une image

Un exemple typique de tableau à deux dimensions est une image bitmap. Chaque élément du tableau représente la couleur associée aux coordonnées. Cette couleur peut être codée de différentes manières :

- Pour une image en noir et blanc, un booléen suffit
- Pour une image en 256 niveaux de gris, on utilise un `unsigned char`
- Pour une image couleur, il est courant d'utiliser trois `unsigned char` qui représentent le rouge, le vert et le bleu ou bien un seul entier qui combine les couleurs grâce à des tranches de bits.
- Il existe encore bien des manières de représenter une couleur (espaces couleurs)

Exemple : une image de taille 640x480 en couleur peut être représentée par :

```
typedef struct {
  unsigned char r;
  unsigned char v;
  unsigned char b;
} PIXEL;
typedef PIXEL IMAGE[640][480];

int i,j;
IMAGE une_image;
PIXEL noir = {0,0,0};
for (i=0;i<640;i++)
  for (j=0;j<480;j++)
     une_image[i][i] = noir;
```

▶ Transformer en un tableau à une dimension

Les tableaux multidimensionnels sont assez difficiles à gérer en C. Il est courant de les transformer en tableaux à une dimension en mettant toutes les lignes sur une seule, les éléments étant les uns à la suite des autres. Exemple, une matrice 3 fois 4 devient un tableau de 12 valeurs :

```
float m[3][4];
float M[12];
```

m[0][0]	m[1][0]	m[2][0]	m[3][0]
m[0][1]	m[1][1]	m[2][1]	m[3][1]
m[0][2]	m[1][2]	m[2][2]	m[3][2]

devient :

M[0]	M[1]	M[2]	…	M[9]	M[10]	M[11]

et on a :

```
m[i][j]  => M[i*3+j]
M[k]     => m[k/3][k%3]
```

Ceci est généralisable pour toutes les dimensions. Cela permet de faire de l'allocation dynamique de zones contiguës plus facilement et augmente les performances.

Questions de synthèse

1. Quelle commande sert à définir un nouveau type ?
2. Peut-t-on définir une structure sans définir de nouveau type ?
3. Quels sont les deux façons d'accéder à un membre d'une structure ?
4. La taille d'une structure est nécessairement plus grande que la somme des tailles de ses membres, vrai ou faux ?
5. Peut-t-on affecter une structure à une autre (de même nature) ?
6. La définition d'une énumération définit un ensemble de constantes entières, vrai ou faux ?
7. Comment peut-t-on déclarer un membre d'une structure comme étant du même type que cette structure ?

Réponses : 1. `typedef` 2. Oui, grâce à `struct` mais sans utiliser `typedef`. 3. Le point . s'il s'agit d'une structure et la flèche -> s'il s'agit d'une pointeur sur une structure. 4. Non, elle peuvent être égales (c'est le cas s'il n'y a aucun alignement). 5. Oui, chaque membre est recopié. 6. Vrai. 7. On ne peut le faire qu'avec un pointeur et en nommant explicitement la structure. Exemple : `typedef struct tag {struct tag * autoreference;} Structure;`

CHAPITRE 9
ALLOCATION DYNAMIQUE

9.1 Introduction

Pour l'instant, on utilise des allocations automatiques faites par le compilateur mais cela présente plusieurs inconvénients :

- On est obligé de se fixer des limites pour les tailles de tableaux (sauf si on utilise la norme c99)
- Une fonction ne peut pas allouer de la mémoire qu'elle retournerait sous forme d'un pointeur utilisable par le contexte appelant
- On peut difficilement faire des structures de données évoluées comme les listes chaînées, les arbres ou les graphes

Tous ces problèmes trouvent une solution dans l'allocation dynamique. Celle-ci consiste à demander au système de réserver de la place mémoire disponible pour le programme. Nous allons étudier la bibliothèque standard permettant de faire cela : `stdlib.h`.

9.2 Allocation

Il existe deux fonctions permettant d'allouer de la mémoire : `malloc` et `calloc`. La première alloue le nombre d'octet passé en paramètre, la deuxième alloue un tableau et met les valeurs (les bits) à zéro :

```
void *malloc(size_t size);
void *calloc(size_t nmemb, size_t size);
```

Ces deux fonctions renvoient un pointeur sur `void` qui correspond à la zone allouée. C'est à l'utilisateur de faire un forçage de type s'il le faut. Lorsque l'allocation ne peut pas être faite, la fonction renvoie NULL (quand il n'y a pas assez de mémoire contigüe disponible par exemple).

▶ Syntaxe type

Le plus souvent, la syntaxe d'une allocation dynamique pour le type `type` ressemblera à cela :

```
type * pointeur;
pointeur = (type*) malloc(sizeof(type)*taille);
```

Voici les différents éléments importants :

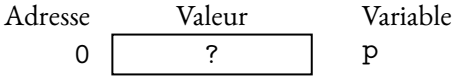
- `(type*)` est un forçage de type pour éviter l'avertissement de compilation. En effet, la fonction `malloc` renvoie le type `void*`.

— `sizeof(type)` permet d'obtenir la taille en octets qu'il faut pour coder le type donné en argument. De cette façon là, on est sûr que la taille est la bonne (elle sera calculée par le compilateur), même s'il s'agit d'un type complexe.
— `taille` est le nombre d'éléments que l'on veut dans le tableau. S'il ne s'agit pas d'un tableau, alors il vaudra 1 (dans ce cas là, on l'omet tout simplement).

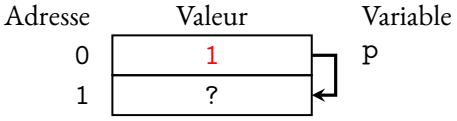
▶ Allocation simple

```
int * p;
p = (int*) malloc(sizeof(int));
*p = 999;
```

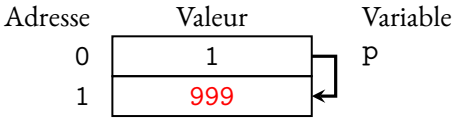
En mémoire avant l'appel à la fonction `malloc` la mémoire du programme ressemble à ceci :



Après l'appel à `malloc`, il s'est passé deux choses :



une nouvelle case mémoire est maintenant utilisable, elle n'a pas de variable directement associée, et son emplacement (1) a été affecté à la variable `p`. Enfin, l'appel à la dernière ligne permet de modifier la valeur de ce nouvel emplacement mémoire en passant par le pointeur :



▶ Allocation de structure

Lorsque l'on veut allouer une structure, on procède exactement de la même manière :

```
typedef struct {int a; int b;} STRUCT;
STRUCT * s;
s = (STRUCT*) malloc(sizeof(STRUCT));
s->a = 1;
s->b = 2;
```

▶ Allocation de tableau

Pour un tableau on utilisera `calloc` ou `malloc` suivant que l'on veut que tous les bits soient mis à zéro ou non :

```
int * tab;
tab = (int*) malloc(sizeof(int)*TAILLE);
```

équivalent à :

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

⚠ Attention, tous les bits sont mis à zéro dans le cas-là, pour les entiers ainsi que les flottants, cela revient à une valeur nulle, mais attention, pour certaines structures, cela peut avoir des significations plus complexes.

9.3 Libération

Dans tout bon programme, à chaque allocation correspond une libération de mémoire. Cela signifie que l'on n'a plus besoin de la zone allouée, le système peut alors l'utiliser pour autre chose. Cette libération n'intervient pas forcément à la fin de l'exécution d'un programme. On fait cette libération grâce à la fonction free :

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

Cette fonction prend tout simplement en paramètre le pointeur renvoyé par l'allocation. Un programme normal doit donc faire ce genre d'opérations :

```
int * p;
p = (int*) malloc(sizeof(int));
...
/* utilisation de p */
...
free(p);
```

Après la libération de la zone allouée, la variable contenant l'adresse existe encore et contient l'ancienne zone allouée. Si vous essayez d'accéder à cette zone, une faute de segmentation risque de se produire. C'est pour cela qu'une bonne habitude de programmation consiste à toujours mettre à NULL une variable après avoir libéré l'adresse qu'elle contenait.

```
free(p);
p = NULL;
```

L'intérêt de l'allocation dynamique est que l'allocation ne se fait pas forcément dans la même fonction que la libération ce qui donne beaucoup plus de possibilités.

9.4 Réallocation

Il existe une fonction pratique qui permet de changer la taille d'une allocation. Elle prend en paramètre l'ancien pointeur et la nouvelle taille, et retourne un nouveau pointeur. Les données présentes à l'ancienne location sont copiées dans la nouvelle.

```
void *realloc(void *ptr, size_t size);
```

Cela peut être utile lorsqu'un tableau doit changer de taille au cours de l'exécution du programme.

⚠ L'argument pointeur passé à realloc doit être un pointeur alloué grâce à malloc, calloc ou realloc, sinon une erreur va se produire. Exemple non valide :

```
int val;
int * p=&val;
p = realloc(p,sizeof(int)*10);
```

se laissera compiler mais provoquera une erreur de segmentation.

9.5 Fonction renvoyant un pointeur

L'intérêt de l'allocation dynamique est que l'on peut faire renvoyer à une fonction un pointeur vers une zone nouvellement allouée. Nous voyons ici quelques exemples classiques.

▶ Création d'un tableau

```
int * new_tableau_int(int taille) {
   int * resultat;
   resultat = (int*) calloc(sizeof(int),taille);
   return resultat;
}
```

qui pourra être utilisé de la manière suivante :

```
int * tab;
tab = new_tableau_int(10);
```

▶ Allocation d'une structure

Le cas de la structure est un peu particulier. En effet, en même temps que l'on alloue la zone nécessaire au stockage de la structure, on veut initialiser les champs qu'elle contient. On peut apparenter cela au constructeur d'un objet. En général, la fonction qui alloue la zone mémoire de la structure prend des paramètres supplémentaires qui doivent renseigner les champs. Exemple : un polygone. Voici les structures de données :

```
typedef struct {
   float x;
   float y;
} POINT;

typedef struct {
   int n;
   POINT * sommets;
} POLYGONE;
```

On déclare deux fonctions, l'une va créer la structure, l'autre la détruire :

```
POLYGONE * new_poly(int n, float * s);
void del_poly(POLYGONE * p);
```

La fonction `new_poly` prend deux arguments :

- Un entier indiquant le nombre de points
- Un tableau de `float` donnant les coordonnées (les unes à la suite des autres) des points

Un exemple d'utilisation :

```
int main(void) {
   POLYGONE * p;
   float sommets[]={0.0,0.0, 1.0,1.0, 0.0,1.0};
   p = new_poly(3,sommets);
   printf("Le périmetre du polygone est %f.\n",perimetre_poly(p));
   del_poly(p);
   return 0;
}
```

Voici la définition des deux fonctions :

```
POLYGONE * new_poly(int n, float * s) {
   POLYGONE * p;
   int i;

   p = (POLYGONE*) malloc(sizeof(POLYGONE));
   p->n = n;
```

```
    p->sommets = (POINT*) malloc(sizeof(POINT)*n);
    for (i=0;i<n;i++)
    {
        p->sommets[i].x = s[i*2];
        p->sommets[i].y = s[i*2+1];
    }
    return p;
}

void del_poly(POLYGONE * p) {
    free(p->sommets);
    free(p);
}
```

Et la fonction supplémentaire permettant le calcul du périmètre :

```
float perimetre_poly(POLYGONE * p) {
    float perimetre=.0,X,Y; int i;
    for (i=0;i<p->n;i++)
    {
        X = p->sommets[i].x - p->sommets[(i+1)%(p->n)].x;
        Y = p->sommets[i].y - p->sommets[(i+1)%(p->n)].y;
        perimetre += sqrt(X*X+Y*Y);
    }
    return perimetre;
}
```

▶ Remarques

Quelques remarques sur ce qui a été évoqué ici :

- On touche à ce que l'on appelle la programmation structurée par types de données. Ceci consiste à définir les structures de données nécessaires ainsi que les opérations que l'on va pouvoir faire dessus : création, destruction, calculs, *etc.*
- Une fonction qui traite une structure de données particulière devra toujours rappeler le nom de cette structure. Dans l'exemple précédent, il s'agit du suffixe `_poly`. De cette manière là, on s'y retrouve facilement. En effet, il est probable que des structures de données aient des fonctions analogues (création, destruction), les appeler `new1`, `new2` ne serait pas très explicite.
- Certains préfèrent utiliser les préfixes plutôt que les suffixes. L'important est de toujours utiliser la même notation (donc de ne pas faire `nouveau_poly` et `poly_detruit` par exemple)
- La programmation structurée par type de données est assez proche de la programmation objet. Vous aborderez cela plus tard dans votre formation.

9.6 Tableaux dynamiques

En allant au delà, on peut imaginer une structure regroupant la taille du tableau et le tableau lui-même. Les opérations que l'on va pouvoir faire dessus sont simples :

- Créer le tableau
- Ajouter un élément
- Obtenir un élément
- Détruire le tableau

Pour optimiser le tout, on décide d'allouer le tableau dynamiquement par morceaux. C'est à dire qu'au fur et à mesure que l'on ajoute des éléments, l'allocation du tableau va être modifiée. Pour ne pas perdre trop de temps en appels systèmes, il ne faut pas allouer à chaque nouvel élément mais tous les PAS nouveaux éléments.
Voici les structures de données :

```
#define PAS 10
typedef int TYPE;
typedef struct {
   int n;       /* nombre effectif d'éléments */
   int max;     /* taille max du tableau */
   TYPE * e;    /* éléments */
} TABLEAU;
```

Les prototypes des fonctions qui vont servir à manipuler le tableau dynamique :

```
TABLEAU * new_tableau(void);
void del_tableau(TABLEAU *tab);
void ajoute_element_tableau(TABLEAU * tab, TYPE elem);
TYPE get_element_tableau(TABLEAU * tab, int i);
```

Un exemple de programme utilisant ces fonctions :

```
int main(void) {
   TABLEAU * tab;
   int i;
   tab = new_tableau();
   for (i=0;i<30;i++)
      ajoute_element_tableau(tab,i);
   printf("Elément no15 = %d.\n",get_element_tableau(tab,15));
   del_tableau(tab);
   return 0;
}
```

Le code des fonctions de construction et destruction :

```
TABLEAU * new_tableau(void) {
   TABLEAU * tab;
   tab = (TABLEAU *) malloc(sizeof(TABLEAU));
   tab->n = 0;
   tab->max = PAS;
   tab->e = (TYPE*) malloc(sizeof(TYPE)*PAS);
   return tab;
}

void del_tableau(TABLEAU *tab) {
   free(tab->e);
   free(tab);
}
```

Le code des fonction d'ajout et d'obtention d'un élément :

```
void ajoute_element_tableau(TABLEAU * tab, TYPE elem) {
   tab->n++;
   if (tab->n>tab->max)
   {
      tab->max += PAS;
      tab->e = realloc(tab->e,sizeof(TYPE)*tab->max);
```

```
    }
    tab->e[tab->n-1] = elem;
}

TYPE get_element_tableau(TABLEAU * tab, int i) {
    return tab->e[i];
}
```

Questions de synthèse

1. Pourquoi fait-t-on un forçage de type sur le retour de la fonction `malloc` ?
2. Quelle fonction est dédiée à l'allocation de tableau ?
3. Le résultat de la fonction `realloc` est-t-il toujours une allocation de mémoire ?
4. Pourquoi faut-t-il libérer la mémoire ?
5. Comment libère-t-on la mémoire ?

Réponses : 1. Car elle renvoie `void*` qui est rarement le type voulu. 2. `calloc` 3. Non, ça peut-être une libération de mémoire. 4. Car cela permet de l'utiliser pour autre chose, de plus certains systèmes ne le font pas systématiquement à la fin de l'exécution d'un programme. 5. Avec la fonction `free`.

CHAPITRE 10
COMPILATION SÉPARÉE

10.1 Fichiers d'en-têtes et d'implémentation

Le langage C différencie la déclaration de la définition de fonctions. Le but est de mettre ceux-ci dans des fichiers différents :

Le fichier d'en-têtes contient toutes les déclarations d'une liste de fonctions. On essaye de regrouper des fonctions ayant des rapports logiques entre elles (par exemple, les fonctions traitant une structure de données). Ce fichier d'en-tête est une sorte de vitrine de la liste de fonctions. S'il est bien documenté, il est suffisant pour un programmeur voulant utiliser les fonctions.

Le fichier d'implémentation contient toutes les définitions associées aux déclarations. Il n'est nécessaire qu'à la personne qui programme son contenu.

▶ Profit

Cette séparation est primordiale pour plusieurs raisons :

— Elle permet de livrer des bibliothèques ne comprenant que le fichier d'en-têtes et le fichier binaire associé donc sans le fichier d'implémentation. Dans un contexte non open-source, c'est extrêmement fréquent.

— Elle permet un travail en groupe productif. Une fois que le fichier d'en-tête est défini, les développeurs peuvent en parallèle programmer du code qui utilise la future bibliothèque et le code à proprement parler de la bibliothèque. Il n'y pas besoin d'attendre que cette bibliothèque soit terminée. On passe d'une méthode séquentielle à une méthode parallèle.

— Elle permet de programmer proprement. Cela implique que les fonctions sont regroupées de manière intelligente (souvent par rapport aux données qu'elles traitent). La structure du programme doit donc être bien pensée à l'avance grâce à une méthode de conception.

▶ Mise en œuvre

En pratique, on utilise l'extension `.h` pour les fichiers d'en-têtes et `.c` pour les fichiers d'implémentation. Le mécanisme `#ifndef FICHIER_H` permet de pouvoir inclure plusieurs fois le fichier d'en-tête sans avoir à se préoccuper des déclarations multiples que cela pourrait occasionner. Imaginez que vous ayez fait une séparation fonctionnelle en deux modules de bibliothèques et que l'un utilise l'autre. Dans le fichiers principal, si vous avez besoin d'utiliser les deux modules, vous serez amenés à faire ce genre de choses :

```
#include "module1.h"
#include "module2.h"
```

Or si le fichier module2.h contient l'inclusion de module1.h, ce dernier sera inclus deux fois. C'est un cas de figure qui n'est pas rare du tout, mieux vaut donc s'en prémunir.

Listing 10.1 – fichier.h

```
#ifndef FICHIER_H
#define FICHIER_H

/* déclarations publiques */
#define CONSTANTE 100
typedef struct {
  ...
} Structure;
int fonction(Structure *,
             double);

#endif
```

Listing 10.2 – fichier.c

```
#include "fichier.h"
/* déclarations privées */
#define CONSTANTE_INTERNE 200

/* définitions */
int fonction(Structure * a,
             double b) {
  /* contenu */
  ...
  return ...;
}
#endif
```

10.2 Fichier principal

Le fichier principal qui utilise les fonctions d'un module doit simplement contenir la clause `#include "module.h"`. Notez la différence avec l'inclusion de fichiers d'en-tête standards (`#include <stdio.h>` par exemple). Une fois que l'inclusion est faite, il devient possible d'utiliser toutes les fonctions déclarées dans le module.

10.3 Phases de compilation séparée

Dans le cas de la compilation séparée, on sépare la compilation de chacun des modules et l'édition des liens. Si on a n modules avec les fichiers correspondants modulei.h et modulei.c, ainsi que le fichier principal principal.c, alors il y aura (n+1) phases de compilation et une phase d'édition des liens. La compilation consiste à transformer un fichier source C en un fichier objet (extension .o sous linux, .obj sous windows). Avec gcc, on utilise l'option -c pour compiler uniquement :

```
gcc -c -o module1.o module1.c
...
gcc -c -o modulen.o modulen.c
gcc -c -o principal.o principal.c
```

Une fois que l'on dispose de tous les fichiers objets, on peut tout rassembler en faisant :

```
gcc -o executable principal.o module1.o ... modulen.o
```

Éventuellement suivi d'un « nettoyage » du fichier qui consiste à enlever toutes les fonctions et symbôles qui ne sont pas utilisées (l'éxécutable est alors plus léger) :

```
strip executable
```

L'éxécutable est maintenant prêt à être utilisé :

```
./executable
```

Remarques :

- À aucun moment on ne fait intervenir les fichiers `.h` dans les lignes de compilations. C'est normal, ils sont automatiquement lus par le préprocesseur lorsqu'une clause `#include` est détectée.
- Si le code source d'un module change, il faut non seulement recompiler ce module mais en plus recompiler les éventuels modules dépendant de lui et refaire l'édition des liens.

10.4 Dépendances et Makefile

Taper les lignes de compilation peut être très contraignant à la longue. L'outil Makefile permet d'automatiser tout ceci. Il existe un autre outil qui s'appelle `makedepend` et qui permet d'automatiser la liste des dépendances de chaque fichier. L'appel à `makedepend` se fait en invoquant des noms de fichiers sources. Exemple :

```
makedepend module.c
```

va analyser le fichier `module.c` et mettre à jour le fichier `Makefile` en y ajoutant une ligne du type :

```
module.o: module.h
```

En fait, `makedepend` implémente un préprocesseur complet qui va aller chercher la liste des fichiers qui sont inclus par un fichier source C. À chaque fois qu'il en trouve un nouveau, une dépendance est créée. Cela signifie que les dépendances entre modules sont automatiquement analysées et vous n'avez plus à vous poser de question sur ce que vous devez recompiler quand vous mettez à jour tel ou tel fichier. Ajoutez simplement la ligne suivante à votre Makefile :

```
depend:
<tab> makedepend *.c
```

Et faites la mise à jour lorsqu'une modification de structure du logiciel intervient avec :

```
make depend
```

Remarque : Dans des environnements de développement intégrés comme Visual C++, les dépendances sont aussi analysées automatiquement (notion de projet).

10.5 Création de bibliothèques

Il est possible de regrouper plusieurs modules dans ce que l'on appelle une bibliothèque statique. L'avantage est que l'on a besoin d'un seul fichier pour faire l'édition des liens. Ce fichier porte l'extension `.a` (comme archive).
Pour créer une archive sous unix :

```
ar rcs bibliotheque.a module1.o module2.o ...
```

Pour utiliser une archive :

```
gcc -o executable archive.a autres_objets.o ...
```

Pour obtenir des informations sur ce que contient une archive :

```
nm archive.a
```

donnera la liste des fichiers `.o` qu'elle contient ainsi que la liste des symboles définis et utilisés dans chacun d'eux.

Questions de synthèse

1. Que contient essentiellement un fichier d'en-têtes ?
2. Peut-t-on compiler directement un fichier d'en-têtes ?
3. `#include "fichier.c"` a-t-il un sens ?
4. Les librairies statiques sous linux ne sont rien d'autre que des fichiers objets mis dans une archive, vrai ou faux ?

Réponses : 1. Des déclarations. 2. Non, ça n'a pas beaucoup de sens. 3. Non. 4. Vrai.

CHAPITRE 11
ÉLÉMENTS DE PROGRAMMATION AVANCÉE

11.1 Les mélanges de types

▶ Forçage de type

On appelle forçage de type l'opération qui consiste à transformer un type en un autre. Bien entendu, il faut que ces types soit compatibles. Lorsqu'ils ne le sont pas, le compilateur affiche un message d'avertissement et le comportement du programme est indéfini. Exemples de forçages de types valides :

```
float x=1.0;
int a;
int * p;
a = (int) x;
x = (float) a;
p = (int*) &x; /* valide mais bizarre ! */
```

▶ Expressions numériques

Le compilateur définit des règles pour le typage des opérations numériques :

- Si tous les arguments sont des entiers, l'opération est entière et le résultat aussi
- Si au moins l'un des arguments est de type flottant, alors tous les autres arguments sont convertis en flottants et l'opération est faite en flottants
- Si plusieurs précisions sont mélangées, on converti tous les arguments à la plus grande des précisions

Ainsi, `5/2` est une opération entière dont le résultat est un entier et vaut donc 2. Par contre, si l'on met `5/2.0`, le résultat sera flottant et vaudra 2.5. L'expression `'A'+10` n'est pas un caractère mais un entier. Le langage C accepte les forçages de types numériques. Ainsi `(int) (2.0*M_PI)` sera de type entier (un arrondi sera effectué). Lorsqu'il y a une affectation qui présente un type différent à droite d'à gauche, le forçage de type est fait automatiquement sans qu'il n'y ait aucun warning.

11.2 Pointeurs de fonctions

En C, on peut définir un pointeur de fonction, qui représente l'adresse d'une fonction ayant un prototype précis. Si la fonction est de ce type :

```
type_retour fonction(type_arg1,type_arg2,...);
```

alors le pointeur sur cette fonction sera du type :

```
type_retour (*pointeur_fonction) (type_arg1,type_arg2,...);
```

Cela permet de passer en argument des pointeurs de fonctions, qui peuvent être appelés à l'intérieur d'une autre fonction.
Exemple, une fonction permet d'afficher un entier, on désire faire une fonction qui affiche un tableau d'entiers :

```
void affiche_int(int a) {
  printf("Valeur de l'entier : %d\n",a);
}
void affiche_tableau_int(int taille,
          int tab[], void (*fonction)(int)) {
  int i;
  printf("Tableau de %d entiers :\n",taille);
  for (i=0;i<taille;i++) fonction(tab[i]);
}
...
int t[] = {1,2,3,4,5};
affiche_tableau_int(5,t,affiche_int);
```

Cela permet de rendre plus modulaire certaines fonctions.

11.3 Directives de préprocesseur

Comme nous l'avons déjà vu, `#include` permet d'inclure un fichier d'en-tête :

- `#include <nom.h>` pour un fichier standard
- `#include "nom.h"` pour un fichier personnel

La commande `#define` permet de définir des macros (avec ou sans arguments). La commande `#ifdef` permet d'exécuter une partie de code seulement si son argument est défini :

```
#define AFFICHE_LONG
#ifdef AFFICHE_LONG
  printf("La valeur vaut %d\n",a);
#else
  printf("%d\n",a);
#endif
```

La commande `#ifndef` permet d'exécuter du code si son argument n'est pas définit :

```
#ifndef bool
#include <stdbool.h>
#endif
```

La commande `#undef` permet d'annuler la définition faite grâce à un `#define`.
Pour tester la définition de plusieurs macros en même temps, on peut utiliser la commande `#if defined(macro)` :

```
#if defined(DEBUG) || defined(VERBOSE)
  printf("...");
#endif
```

11.4 Récursivité

La récursivité s'implémente de manière simple en C. Il suffit de faire appel à la fonction dans laquelle on se trouve. Bien-sûr il faudra soigner la condition d'arrêt et veiller à effectuer les opérations avant ou après le dépilement suivant le cas de figure.

▶ Exemple simple

L'exemple le plus classique est celui du calcul de la factorielle d'un nombre entier. La fonction factorielle peut être définie par la relation de récurrence suivante :

$$\begin{cases} (n)! & = & (n)(n-1)! \ \forall n > 0 \\ 0! & = & 1 \end{cases}$$

La traduction en C reprend exactement cette définition :

```
int factorielle(int n) {
    if (n==0)
        return 1;
    else
        return n*factorielle(n-1);
}
```

Simulons son exécution pour factorielle(4) :

```
factorielle(4) -> n!=0 -> return 4*factorielle(3)
  factorielle(3) -> n!=0 -> return 3*factorielle(2)
    factorielle(2) -> n!=0 -> return 2*factorielle(1)
      factorielle(1) -> n!=0 -> return 1*factorielle(0)
        factorielle(0) -> n==0 -> return 1
      return 1*1 = 1
    return 2*1 = 2
  return 3*2 = 6
return 4*6 = 24
```

Les empilement d'appels sont représentés par une indentation et les dépilements par une indentation négative. C'est une bonne habitude de vérifier le fonctionnement d'une fonction récursive à la main.

▶ Exemple plus complexe

La récursivité pose plus de problèmes lorsque certaines opérations sont faites avant ou après le dépilement :

— Des opérations sont faites avant le dépilement lorsqu'elles interviennent avant l'appel récursif
— Des opérations sont faites après le dépilement lorsqu'elles interviennent après l'appel récursif

Exemple flagrant :

```
#include <stdio.h>

void affiche_avant(char * chaine) {
    if (*chaine != '\0') {
        putchar(*chaine);
        affiche_avant(chaine+1);
    }
}
void affiche_apres(char * chaine) {
    if (*chaine != '\0') {
        affiche_apres(chaine+1);
        putchar(*chaine);
    }
}
```

```
int main(void) {
    affiche_avant("Coucou");
    printf("\n");
    affiche_apres("Coucou");
    printf("\n");
    return 0;
}
```

Affichera à l'exécution :

Coucou
uocuoC

Dans le premier cas, on affiche avant le dépilement, donc dans l'ordre d'appel. Dans le deuxième cas, on affiche après le dépilement, donc dans l'ordre inverse des appels successifs.

11.5 Liste chaînée

Pour implémenter une liste chaînée, il suffit de définir une structure qui s'autoréférence :

```
typedef struct tag_liste {
  Valeur valeur;
  struct tag_liste * suivant;
} Liste;
```

Ainsi, on définit une liste comme étant le pointeur sur le premier élément de la liste :

```
Liste * liste;
```

Par convention, il est pratique d'initialiser cette liste à NULL pour indiquer qu'elle ne contient aucun élément.

⚠ La fonction qui ajoute un élément à la liste doit pouvoir modifier la racine donc son prototype sera au choix :

```
Liste * insere_element(Liste * racine,Valeur valeur);
void insere_element(Liste ** racine,Valeur valeur);
```

Dans le premier cas, on l'utilisera ainsi :

```
liste = insere_element(liste, v);
```

Dans le deuxième :

```
insere_element(&liste,v);
```

Il en est de même pour la suppression.
Exemple de fonction parcourant la liste chaînée et appliquant une fonction à chaque valeur contenue dans les éléments (version récursive) :

```
void parcours_liste(Liste * racine,
                    void (*fonction)(Valeur)) {
  if (racine != NULL) {
    fonction(racine->valeur);
    parcours_liste(racine->suivant,fonction);
  }
}
```

Version non récursive (moins gourmande en mémoire) :

```
void parcours_liste(Liste * racine,
                    void (*fonction)(Valeur)) {
  while (racine != NULL) {
    fonction(racine->valeur);
    racine = racine->suivant;
  }
}
```

11.6 Arbre

Un arbre est souvent implémenté de la façon suivante :

```
#define VALENCE 2
typedef struct tag_arbre {
  Valeur valeur;
  struct tag_arbre * branches[VALENCE];
} Arbre;
Arbre * racine;
```

Il est courant de mettre la valeur NULL aux branches lorsqu'il n'y en a pas (cas d'une feuille). Comme pour les listes chaînées, il faudra garder à l'esprit que la racine peut être modifiée lorsque l'arbre passe de l'état de vide à non-vide (et inversement).
Exemple de fonction parcourant l'arbre récursivement :

```
void parcours_arbre(Arbre * racine,
                    void (*fonction)(Valeur)) {
  if (racine != NULL) {
    fonction(racine->valeur);
    parcours_arbre(racine->branches[0]);
    parcours_arbre(racine->branches[1]);
  }
```

Dans ce cas là, un parcours non récursif n'est pas possible.

11.7 Fichiers binaires

Un fichier binaire est un fichier structuré dans lequel on peut accéder aux enregistrements sans lire tout ce qui se trouve avant. Ceci rend possible la modification en même temps que la lecture. En général, on définit une structure qui va correspondre à un enregistrement.

▶ Ouverture du fichier

Il faut utiliser bien entendu la fonction `fopen`, mais en prenant soin de mettre un `+` pour spécifier que l'on veut faire de la lecture et de l'écriture et le `b` pour indiquer que l'on travaille en binaire (sous les systèmes POSIX comme linux, cela n'a aucune influence, mais c'est bon de le mettre pour des raisons de portabilité). Exemple d'ouverture de fichier existant (pas de création) :

```
if ((fid=fopen("monfichier.bin","rb+"))==NULL)
{
  fprintf(stderr,"Erreur à l'ouverture du fichier\n");
  exit(1);
}
```

▶ Positionnement dans le fichier

Chaque enregistrement contient l'équivalent d'une structure `Structure`. Le premier enregistrement se trouve donc à la position 0, le deuxième à la position `sizeof(Structure)`, le troisième à `2*sizeof(Structure)`, *etc.* Si `taille=sizeof(Structure)`, alors :

Position dans le fichier	0	taille	2*taille	*etc.*
Numéro d'enregistrement	0	1	2	...

Que ce soit avant la lecture ou avant l'écriture, il faut se positionner dans le fichier avant d'effectuer l'opération voulue. On utilise pour cela la fonction `fseek` :

```
int fseek( FILE *stream, long offset, int whence);
```

Le premier argument est le flux, le deuxième le décalage que l'on désire, et le troisième indique le repère :
- `SEEK_SET` pour un positionnement absolu (par rapport au début du fichier donc)
- `SEEK_CUR` pour un positionnement relatif (par rapport à la position courante)
- `SEEK_END` pour un positionnement par rapport à la fin du fichier (on doit alors spécifier une valeur de décalage négative)

Pour atteindre l'enregistrement `n`, on procédera de la manière suivante :

```
resultat = fseek(flux,n*sizeof(Structure),SEEK_SET);
```

La valeur de retour de `fseek` vaut 0 si tout s'est bien passé, -1 sinon.

▶ Lecture d'un enregistrement

La lecture d'un ou de plusieurs enregistrements se fait grâce à la fonction `fread` :

```
size_t fread( void *ptr, size_t size, size_t nmemb, FILE *stream);
```

Le premier argument est l'adresse à laquelle va être stockée le contenu du fichier lu (typiquement de type `Structure *`, le deuxième indique la taille d'un enregistrement, le troisième le nombre d'enregistrements et le dernier le flux de fichier. Exemple :

```
Structure element;
fread(&element,sizeof(Structure),1,flux);
```

Exemple de lecture de plusieurs enregistrements :

```
Structure elements[10];
fread(elements,sizeof(Structure),10,flux);
```

La fonction `fread` renvoie le nombre d'éléments correctement lus (peut être inférieur au nombre d'élément spécifié en argument si on atteint la fin du fichier par exemple).

▶ Écriture d'un enregistrement

Il s'agit du même principe, mais cette fois-ci, les données pointées par le premier argument sont lues et écrites dans le fichier :

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

Exemple d'écriture d'un enregistrement :

```
Structure element = ...;
fwrite(&element,sizeof(Structure),1,flux);
```

⚠ Bien que `element` ne soit pas modifié, il y a quand même un passage de l'adresse pour plus de légèreté dans le passage d'argument.

▶ Connaître la taille d'un fichier

Il n'existe pas de fonction faisant cette tache directement. Il faut en utiliser plusieurs. L'astuce est de se positionner à la fin du fichier et de demander la position courante avec la fonction `ftell`. Voici un petit utilitaire qui vous permettra de tout faire. Il retourne -1 si le fichier n'existe pas. Il retournera 0 si le fichier existe mais qu'il est vide.

```
long taille_fichier(char * nom) {
   FILE * flux;
   long taille;
   flux = fopen(nom,"r");
   if (!flux)
      return -1;
   if (fseek(flux,0L,SEEK_END)==-1) {
      fclose(flux);
      return -1;
   }
   taille = ftell(flux);
   fclose(flux);
   return taille;
}
```

Ainsi, on peut connaître le nombre d'enregistrements en divisant par la taille d'un enregistrement :

```
n_enreg = taille_fichier(nomfichier)/sizeof(Structure);
```

11.8 Polymorphisme

On parle de polymorphisme en programmation orientée objet lorsqu'une fonction (ou plutôt une méthode) est capable de traiter plusieurs instances d'objets différents et que son comportement dépend du type de l'objet. En C, ce mécanisme n'est pas automatique, mais il est possible de le reproduire assez facilement. Pour cela on construit des structures adaptées :

```
typedef struct {
   int type;
} Objet;
typedef struct {
   Objet o;
   int v;
   float x;
} Objet1;
typedef struct {
   Objet o;
   char chaine[10];
} Objet2;
```

La structure `Objet` sert de base aux autres, elle ne contient qu'un champs qui correspond à un indicateur de type (c'est un entier). Les deux autres structures ont comme premier membre obligatoirement la structure `Objet`, c'est ce qui va permettre de les différencier. Voici l'exemple d'une fonction qui prend comme argument un pointeur sur `Objet`, qui analyse le contenu du premier champs et qui fait des opérations en fonction de celui-ci :

```
void affiche_elements(Objet * o) {
   switch (o->type) {
      case 1:
```

```
            printf("Objet de type 1 : %d, %f\n",
                ((Objet1*)(o))->v,((Objet1*)(o))->x);
            break;
        case 2:
            printf("Objet de type 2 : %s\n",
                ((Objet2*)(o))->chaine);
            break;
    }
}
```

Voici un exemple d'utilisation :

```
int main(void) {
    Objet1 o1 = {{1},2,3.5};
    Objet2 o2 = {{2},"Coucou"};
    affiche_elements(&o1);
    affiche_elements(&o2);
    return 0;
}
```

qui affichera :

```
Objet de type 1 : 2, 3.500000
Objet de type 2 : Coucou
```

- ⚠ La fonction doit forcément prendre comme argument l'adresse de la structure, car quand le type de structure est différent, sa taille varie, on ne passerait sur la pile qu'une partie de la structure.
- ⚠ Le premier champs des deux structures `Objet1` et `Objet2` doit obligatoirement être la structure `Objet` (sans pointeur) car c'est ce qui permet de faire `switch (o->type)` en atteignant directement le champs type (alors qu'il n'existe pas dans les deux structures).
- ⚠ Ce code génère des warnings à la compilation, c'est normal, la fonction attend un type `Objet*` et elle reçoit `Objet1*`. C'est justement l'intérêt du code, donc ce n'est pas grave !

Questions de synthèse

1. Le C est capable de faire des conversions de type automatiquement, vrai ou faux ?
2. On peut passer l'adresse d'une fonction à une autre fonction, vrai ou faux ?
3. Un arbre peut-t-il être parcouru de façon récursive ?
4. Existe-t-il en C une fonction standard permettant de connaître la taille d'un fichier ?

Réponses : 1. Vrai, pour les types numériques par exemple. 2. Vrai, grâce à une pointeur de fonction. 3. Oui, c'est d'ailleurs la seule façon. 4. Non, il faut l'ouvrir et se positionner à la fin pour en connaître la taille.

ANNEXE A
FICHE DE SYNTHÈSE

Cette fiche récapitule les principaux éléments syntaxiques du langage C.

A.1 Préprocesseur

Inclusion d'un fichier

```
#include <fichier_systeme.h>
#include "fichier_perso.h"
```

Définition d'une constante

```
#define NOM Valeur
```

Définition d'une macro

```
#define macro(x) (fonction de x)
```

Clause conditionnelle

```
#ifdef identificateur
ou #ifndef identificateur
ou #if expression_constante
...
#else
...
#endif
```

A.2 Déclarations

Elles sont toujours terminées par un point-virgule.
Une variable

```
type_var nom_var1, nom_var2, ...;
```

Une fonction

```
type_retour nom_fonction(type_par1,
                         type_par2, ...);
```

Un tableau

```
type_element nom_tableau[nb_elements];
```

Un pointeur

```
type_variable * nom_pointeur;
```

A.3 Structures, énumérations, unions

Structure

```
struct nom_structure {
  type_membre1 nom_membre1;
  type_membre2 nom_membre2;
  ...;
};
```

Structure avec typedef

```
typedef struct nom_structure {
  type_membre1 nom_membre1;
  type_membre2 nom_membre2;
  ...;
} type_structure;
```

Énumération

```
enum nom_enumeration {
  mnemonique1, mnemonique2, ...
};
```

Énumération avec typedef

```
typedef enum nom_enumeration {
  mnemonique1, mnemonique2, ...
} type_enumeration;
```

A.4 Opérateurs

Arithmétiques

```
+ - / * %
```

Unaires

```
++ -- ! & sizeof (type)
```

Relationnels

```
< <= > >= == !=
```

Logiques

```
&& || !
```

Bit à bit

```
& | ^ << >>
```

Opérateurs avec affectation

```
= += -= *= /= %=
```

Opérateur conditionnel

```
expr1?expr2:expr3
```

A.5 Structures de contrôle

Tantque ... faire ...

```
while (expression)
{
  ...;
}
```

Faire ... tantque ...

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

Pour

```
for (expr_init;cond_continuer;expr_iter)
{
  ...;
}
```

Si ... alors ...

```
if (condition)
{
  ...;
}
```

Si ... alors ... sinon ...

```
if (condition)
{
  ...;
}
else
{
  ...;
}
```

Suivant la valeur de ... faire ...

```
switch (expression)
{
  case valeur1:
    ...;
    [break;]
  case valeur2:
    ...;
    [break;]
  default:
    ...;
}
```

ANNEXE B
RÉFÉRENCES DE FONCTIONS

Vous trouverez ici des extraits de pages de manuel linux (traduction française). Pour plus de détails sur une fonction précise, tapez `man fonction`.

B.1 Entrées-sorties

Les fonctions d'entrée-sorties se font par l'intermédiaire de la bibliothèque `<stdio.h>`.

---

fopen — Ouverture d'un fichier — fopen

Prototype `#include <stdio.h>`
`FILE *fopen (const char *path, const char *mode);`

Description La fonction fopen ouvre le fichier dont le nom est contenu dans la chaine pointée par path et lui associe un flux.

L'argument mode pointe vers une chaîne commençant par l'une des séquences suivantes (d'autres caractères peuvent suivre la séquence) :

r Ouvre le fichier en lecture. Le pointeur de flux est placé au début du fichier.

r+ Ouvre le fichier en lecture et écriture. Le pointeur de flux est placé au début du fichier.

w Ouvre le fichier en écriture. Le fichier est créé s'il n'existait pas. S'il existait déjà, sa longueur est ramenée à 0. Le pointeur de flux est placé au début du fichier.

w+ Ouvre le fichier en lecture et écriture. Le fichier est créé s'il n'existait pas. S'il existait deja, sa longueur est ramenée à 0. Le pointeur de flux est placé au début du fichier.

a Ouvre le fichier en écriture. Le fichier est créé s'il n'existait pas. Le pointeur de flux est placé à la fin du fichier.

a+ Ouvre le fichier en lecture et écriture. Le fichier est créé s'il n'existait pas. Le pointeur de flux est placé à la fin du fichier.

Valeur renvoyée S'il réussit intégralement fopen renvoie un pointeur sur un fichier, de type FILE. Sinon, il renvoie NULL et errno contient le code d'erreur.

---

fclose — Fermeture d'un fichier — fclose

Prototype
```
#include <stdio.h>
int fclose (FILE *stream);
```

Description La fonction fclose dissocie le flux nommé stream du fichier sous-jacent. Si le flux était utilisé en sortie, toutes les données contenues dans le buffer sont d'abord écrites, en utilisant fflush(3) .

Valeur renvoyée Si la fonction réussit intégralement, elle renvoie 0, sinon elle renvoie EOF et errno contient le code d'erreur. Dans tous les cas, tout autre accès ultérieur au flux (y compris un autre appel de fclose()) conduit à un comportement indéfini.

---

fread, fwrite — Entrées/sorties binaires sur un flux — fread, fwrite

Prototype
```
#include <stdio.h>
size_t fread (void *ptr,
              size_t size,
              size_t nmemb,
              FILE *stream);
size_t fwrite (const void *ptr,
               size_t size,
               size_t nmemb,
               FILE *stream);
```

Description La fonction fread lit nmemb éléments de données, chacun d'eux représentant size octets de long, depuis le flux pointé par stream, et les stocke à l'emplacement pointé par ptr.

La fonction fwrite écrit nmemb éléments de données, chacun d'eux représentant size octet de long, dans le flux pointé par stream, après les avoir lus depuis l'emplacement pointé par ptr.

Valeur renvoyée fread et fwrite renvoient le nombre d'éléments correctement lus ou écrits (et non pas le nombre d'octets). Si une erreur se produit, ou si la fin du fichier est atteinte en lecture, le nombre renvoyé est plus petit que nmemb et peut même être nul.

fread traite la fin du fichier comme une erreur, et l'appelant devra appeler feof(3) ou ferror(3) pour distinguer ce cas.

---

fgetc, getc, getchar — Lecture-écriture de caractères — fgetc, getc, getchar

Prototype
```
#include <stdio.h>
int fgetc (FILE *stream);
int getc (FILE *stream);
int getchar (void);
```

Description fgetc() lit le caractère suivant depuis le flux stream et renvoie ce caractère, lu sous forme unsigned char, puis transformé en int, ou EOF en cas d'erreur ou de fin de fichier. getc() est équivalent à fgetc() sauf qu'il peut être implementé sous forme de macro, qui évalue l'argument stream plusieurs fois. getchar() est équivalent à getc(stdin).

Valeur renvoyée fgetc(), getc() et getchar() renvoie un caractère, lu comme un unsigned char, et transformé en int, ou EOF à la fin du fichier, ou en cas d'erreur.

fputc, putc, putchar — Écriture de caractères — fputc, putc, putchar

Prototype
```
#include <stdio.h>
int fputc (int c, FILE *stream);
int putc (int c, FILE *stream);
int putchar (int c);
```

Description fputc() écrit le caractère c, transformé en unsigned char, dans le flux stream. putc() est équivalent à fputc() sauf qu'il peut être implémenté comme une macro qui évalue plusieurs fois son argument stream. putchar(c) est équivalent à putc(c,stdout).

Valeur renvoyée fputc(), putc() et putchar() renvoient le caractère écrit en tant qu'unsigned char, converti en int ou EOF en cas d'erreur.

fgets, fputs — Lecture/écriture de chaînes — fgets, fputs

Prototype
```
#include <stdio.h>
int fputs (const char *s, FILE *stream);
int puts (const char *s);
char *fgets (char *s, int size, FILE *stream);
char *gets (char *s);
```

Description fputs() écrit la chaîne de caractères s dans le flux stream, sans écrire le '\0' final. puts() écrit la chaîne de caractères s et un retour-chariot final sur stdout.

fgets() lit au plus size - 1 caractères depuis stream et les place dans le buffer pointé par s. La lecture s'arrête après EOF ou un retour-chariot. Si un retour-chariot (newline) est lu, il est placé dans le buffer. Un caractère nul '\0' est placé à la fin de la ligne.

gets() lit une ligne depuis stdin et la place dans le buffer pointé par s jusqu'à atteindre un retour-chariot, ou EOF, qu'il remplace par '\0'. Il n'y a pas de vérification de débordement de buffer.

Valeur renvoyée

printf, etc. — Sorties formatées — printf, etc.

Prototype
```
#include <stdio.h>
int printf (const char *format, ...);
int fprintf (FILE *stream, const char *format, ...);
int sprintf (char *str, const char *format, ...);
int snprintf (char *str, size_t size, const char *format, ...);
```

Description Les fonctions de la famille printf produisent des sorties en accord avec le format décrit plus bas. La fonction printf écrit sur stdout, le flux de sortie standard. fprintf écrit sur le flux stream indiqué. sprintf, et snprintf écrivent leurs sorties dans la chaîne de caractères str.

Ces quatre fonctions créent leurs sorties sous le contrôle d'une chaîne de format qui indique les conversions à apporter aux arguments suivants

Valeur renvoyée Ces fonctions renvoient le nombre de caractères imprimés, sans compter le caractère nul '\0' final dans les chaînes. snprintf n'écrit pas plus de size octets (y compris le '\0' final), et

renvoie -1 si la sortie a été tronquée à cause de cette limite. Ceci était vrai jusqu'à la GlibC 2.0.6. Depuis la GlibC 2.1, ces fonctions suivent le standard C99 et renvoient le nombre de caractères (sans compter le '\0' final) qui auraient été écrits si la chaîne à remplir avait été assez longue.

Chaîne de format Le format de conversion est indiqué par une chaîne de caractères, commençant et se terminant dans son état de décalage initial. La chaîne de format est composée d'indicateurs : les caractères ordinaires (différents de %), qui sont copiés sans modification sur la sortie, et les spécifications de conversion, qui sont mises en correspondances avec les arguments suivants. Les spécifications de conversion sont introduites par le caractère %, et se terminent par un indicateur de conversion. Entre eux peuvent se trouver (dans l'ordre), zéro ou plusieurs attributs, une valeur optionnelle de largeur minimal de champ, une valeur optionnelle de précision, et un éventuel modificateur de longueur.

Attribut Le caractère % peut être éventuellement suivi par un ou plusieurs attributs suivants :

- # indique que la valeur doit être convertie en une autre forme. Pour la conversion o le premier caractère de la chaîne de sortie vaudra zéro (en ajoutant un préfixe o si ce n'est pas déjà un zéro). Pour les conversions x et X une valeur non nulle reçoit le préfixe '0x' (ou '0X' pour l'indicateur X). Pour les conversions a, A, e, E, f, g, et G le résultat contiendra toujours un point décimal même si aucun chiffre ne le suit (normalement, un point décimal n'est présent avec ces conversions que si des décimales le suivent). Pour les conversions g et G les zéros en tête ne sont pas éliminés, contrairement au comportement habituel. Pour les autres conversions, cet attribut n'a pas d'effet.
- 0 indique le remplissage avec des zéros. Pour les conversions d, i, o, u, x, X, a, A, e, E, f, F, g, et G, la valeur est complétée à gauche avec des zéros plutot qu'avec des espaces. Si les attributs 0 et - apparaissent ensemble, l'attribut 0 est ignoré. Si une précision est fournie avec une conversion numérique (d, i, o, u, x, et X), l'attribut 0 est ignoré. Pour les autres conversions, le comportement est indéfini.
- - indique que la valeur doit être justifiée sur la limite gauche du champ (par défaut elle l'est à droite). Sauf pour la conversion n, les valeurs sont complétées à droite par des espaces, plutôt qu'a gauche par des zéros ou des blancs. Un attribut - surcharge un attribut 0 si les deux sont fournis.
- ' ' (un espace) indique qu'un espace doit être laissé avant un nombre positif (ou une chaîne vide) produit par une conversion signée
- + indique que le signe doit toujours être imprimé avant un nombre produit par une conversion signée. Un attribut + surcharge un attribut 'espace' si les deux sont fournis.

Largeur de champ Un nombre optionnel ne commençant pas par un zéro, peut indiquer une largeur minimale de champ. Si la valeur convertie occupe moins de caractères que cette largeur, elle sera complétée par des espaces à gauche (ou à droite si l'attribut d'alignement à gauche a été fourni). À la place de la chaîne représentant le nombre décimal, on peut écrire '*' ou '*m$' (m étant entier) pour indiquer que la largeur du champ est fournie dans l'argument suivant, ou dans le m-ième argument, respectivement. L'argument fournissant la largeur doit être de type int. Une largeur négative est considéré comme l'attribut '-' vu plus haut suivi d'une largeur positive. En aucun cas une largeur trop petite ne provoque la troncature du champ. Si le résultat de la conversion est plus grand que la largeur indiquée, le champ est élargi pour contenir le résultat.

Précision Une précision eventuelle, sous la forme d'un point ('.') suivi par un nombre. À la place de la chaîne représentant le nombre décimal, on peut écrire '*' ou '*m$' (m étant entier) pour indiquer que la précision est fournie dans l'argument suivant, ou dans le m-ième argument, respectivement. L'argument fournissant la précision doit être de type int. Si la précision ne contient

que le caractère '.', ou une valeur négative, elle est considérée comme nulle. Cette précision indique un nombre minimum de chiffres à faire apparaître lors des conversions d, i, o, u, x, et X, le nombre de décimales à faire apparaître pour les conversions a, A, e, E, f, et F, le nombre maximum de chiffres significatifs pour g et G, et le nombre maximum de caractères à imprimer depuis une chaîne pour les conversions S et s.

Modificateur de longueur Ici, une conversion entière correspond à d, i, o, u, x, ou X.

hh La conversion entière suivante correspond à un signed char ou unsigned char, ou la conversion n suivante correspond à un argument pointeur sur un signed char.

h La conversion entière suivante correspond à un short int ou unsigned short int, ou la conversion n suivante correspond à un argument pointeur sur un short int.

l La conversion entière suivante correspond à un long int ou unsigned long int, ou la conversion n suivante correspond à un pointeur sur un long int, ou la conversion c suivante correspond à un argument wint_t, ou encore la conversion s suivante correspond à un pointeur sur un wchar_t.

ll La conversion entière suivante correspond à un long long int, ou unsigned long long int, ou la conversion n suivante correspond à un pointeur sur un long long int.

L La conversion a, A, e, E, f, F, g, ou G suivante correspond à un argument long double.

j La conversion entière suivante correspond à un argument intmax_t ou uintmax_t.

z La conversion entière suivante correspond à un argument size_t ou ssize_t. (La bibliothèque libc5 de Linux proposait l'argument Z pour cela, ne pas utiliser).

t La conversion entière suivante correspond à un argument ptrdiff_t.

Indicateur De Conversion Un caractère indique le type de conversion à apporter. Les indicateurs de conversion, et leurs significations sont :

d, i L'argument int est convertie en un chiffre décimal signé. La précision, si elle est mentionnée, correspond au nombre minimal de chiffres qui doivent apparaître. Si la conversion fournit moins de chiffres, le résultat est rempli à gauche avec des zéros. Par défaut la précision vaut 1. Lorsque o est converti avec une précision valant 0, la sortie est vide.

o, u, x, X L'argument unsigned int est converti en un chiffre octal non-signé (o), un chiffre décimal non-signé (u), un chiffre héxadécimal non-signé (x et X). Les lettres abcdef sont utilisées pour les conversions avec x, les lettres ABCDEF sont utilisées pour les conversions avec X. La précision, si elle est indiquée, donne un nombre minimal de chiffres à faire apparaître. Si la valeur convertie nécessite moins de chiffres, elle est complétée à gauche avec des zéros. La précision par défaut vaut 1. Lorsque o est converti avec une précision valant 0, la sortie est vide.

e, E L'argument réel, de type double, est arrondi et présenté avec la notation scientifique [-]c.ccce± cc dans lequel se trouve un chiffre avant le point, puis un nombre de décimales égal à la précision demandée. Si la précision n'est pas indiquée, l'affichage contiendra 6 décimales. Si la précision vaut zéro, il n'y a pas de point décimal. Une conversion E utilise la lettre E (plutôt que e) pour introduire l'exposant. Celui-ci contient toujours au moins deux chiffres. Si la valeur affichée est nulle, son exposant est 00.

f, F L'argument réel, de type double, est arrondi, et présenté avec la notation classique [-]ccc.ccc, où le nombre de décimales est égal à la précision réclamée. Si la précision n'est pas indiquée, l'affichage se fera avec 6 décimales. Si la précision vaut zéro, aucun point n'est affiché. Lorsque le point est affiché, il y a toujours au moins un chiffre devant.

g, G L'argument réel, de type double, est converti en style f ou e (ou E pour la conversion G). La précision indique le nombre de décimales significatives. Si la précision est absente, une valeur par défaut de 6 est utilisée. Si la précision vaut 0, elle est considérée comme valant 1. La notation scientifique e est utilisée si l'exposant est inférieur à -4 ou supérieur ou égal à la précision démandée. Les zéros en fin de partie décimale sont supprimés. Un point decimal n'est affiché que s'il est suivi d'au moins un chiffre.

a, A Pour la conversion a, l'argument de type double est transformé en notation hexadécimale (avec les lettres abcdef) dans le style [-]oxh.hhhhp± d ; Pour la conversion A, le préfixe 0X, les lettres ABCDEF et le séparateur d'exposant P sont utilisés. Il y a un chiffre hexadécimal avant la virgule, et le nombre de chiffres ensuite est égal à la précision. La précision par défaut suffit pour une représentation exacte de la valeur, si une représentation exacte est possible en base 2. Sinon elle est suffisament grande pour distinguer les valeurs de type double. Le chiffre avant le point décimal n'est pas spécifié pour les nombres non-normalisés, et il non-nul pour les nombres normalisés.

c S'il n'y a pas de modificateur l, l'argument entier, de type int, est converti en un unsigned char, et le caractère correspondant est affiché. Si un modificateur l est présent, l'argument de type wint_t (caractère large) est converti en séquence multi-octet par un appel à wctomb, avec un état de conversion débutant dans l'état initial. La chaîne multi-octet résultante est écrite.

s S'il n'y a pas de modificateur l, l'argument de type const char * est supposé être un pointeur sur un tableau de caractères (pointeur sur une chaîne). Les caractères du tableau sont écrits jusqu'au caractère NUL final, non compris. Si une précision est indiquée, seul ce nombre de caractères est écrit. Si une précision est fournie, il n'y a pas besoin de caractère nul. Si la précision n'est pas donnée, ou si elle est supérieure à la longueur de la chaîne, le caractère NUL final est nécessaire.

Si un modificateur l est présent, l'argument de type const wchar_t * est supposé être un pointeur sur un tableau de caractères larges. Les caractères larges du tableau sont convertis en une séquence de caractères multi-octets (chacun par un appel de wctomb, avec un état de conversion dans l'état initial avant le premier caractère large), ceci jusqu'au caractère large nul final compris. Les caractères multi-octets résultants sont écris jusqu'à l'octet nul final (non compris). Si une précision est fournie, il n'y a pas plus d'octets écrits que la précision indiquée, mais aucun caractère multi-octet n'est écrit partiellement. Remarquez que la précision concerne le nombre d'octets écrits, et non pas le nombre de caractères larges ou de positions d'écrans. La chaîne doit contenir un caractère large nul final, sauf si une précision est indiquée, suffisament petite pour que le nombre d'octets écrits la remplisse avant la fin de la chaîne.

n Le nombre de caractères déjà écrits est stocké dans l'entier indiqué par l'argument pointeur de type int *. Aucun argument n'est converti.

% Un caractère % est écrit. Il n'y a pas de conversion. L'indicateur complet est %%.

---

scanf, etc. ENTRÉES FORMATÉES scanf, etc.

Prototype `#include <stdio.h>`

```
int scanf (const char *format, ...);
int fscanf (FILE *stream, const char *format, ...);
int sscanf (const char *str, const char *format, ...);
```

Description Les fonctions de la famille scanf analysent leurs entrées conformément au format décrit plus bas. Ce format peut contenir des indicateurs de conversion. Les résultats des conversions,

s'il y en a, sont stockés dans des arguments pointeurs. La fonction scanf lit ses données depuis le flux d'entrée standard stdin, fscanf lit ses entrées depuis le flux pointé par stream, et sscanf lit ses entrées dans la chaîne de caractères pointée par str.

Les arguments pointeurs successifs doivent correspondre correctement aux indicateurs de conversion fournis (voir néanmoins l'attribut '*' plus bas). Toutes les conversions sont introduites par le caractère % (symbole pourcent). La chaîne format peut également contenir d'autres caractères. Les blancs (comme les espaces, les tabulations ou les retours chariots) dans la chaîne format correspondent à un nombre quelconque de blancs (et même aucun) dans la chaine d'entrée. Tous les autres caractères ne peuvent correspondre qu'à eux-même. L'examen de l'entrée s'arrête dès qu'un caractère d'entrée ne correspond pas à un caractère du format. L'examen s'arrête également quand une conversion d'entrée est impossible (voir ci-dessous).

Valeur renvoyée Ces fonctions renvoient le nombre d'éléments d'entrées correctement assignés. Ce nombre peut être plus petit que le nombre d'éléments attendus, et même être nul, s'il y a une erreur de mise en correspondance. La valeur zéro indique qu'aucune conversion n'a été faite bien que des caractères étaient disponibles en entrée. Typiquement c'est un caractère d'entrée invalide qui en est la cause, par exemple un caractère alphabétique dans une conversion %d. La valeur EOF est renvoyée si une erreur d'entrée a eu lieu avant toute conversion, par exemple une fin de fichier. Si une erreur fin-de-fichier se produit après que les conversions aient commencé, le nombre de conversions réussies sera renvoyé.

Conversions A la suite du caractère % introduisant une conversion, il peut y avoir un nombre quelconque de caractères attributs de la liste suivante :

* Ne pas stocker le résultat. La conversion est bien effectuée comme d'habitude, mais le resultat est éliminé au lieu d'être memorisé dans un pointeur.

a Indique que la conversion sera de type s, la mémoire nécessaire pour la chaine sera allouée avec malloc(3) et le pointeur sera assigné à la variable de type char qui n'a pas besoin d'être initialisée auparavant. Cet attribut n'existe pas en C ANSI.

h Indique que la conversion sera de type dioux ou n et que le pointeur suivant est un pointeur sur un short int (plutôt que sur un int).

l Indique que la conversion sera de type dioux ou n et que le pointeur suivant est un pointeur sur un long int (plutôt que sur un int), ou que la conversion sera de type efg et que le pointeur suivant est un pointeur sur un double (plutôt que sur un float). Indiquer deux attributs l successifs est équivalent à indiquer l'attribut L.

L Indique que la conversion sera de type efg et que le pointeur suivant est un pointeur sur un long double ou que la conversion sera de type dioux et que le pointeur suivant est un pointeur sur un long long. (ce type n'existe pas en C ANSI. Un programme l'utilisant ne sera pas portable sur toutes les machines).

q est équivalent à L. Cet attribut n'existe pas en C ANSI.

En plus de ces attributs peut se trouver un champ optionnel de longueur maximale, exprimée sous forme d'entier, entre le caractère % et l'indicateur de conversion. Si aucune longueur n'est donnée, une valeur infinie est utilisée par défaut (avec une exception, voir plus bas). Autrement, la conversion examinera au plus le nombre de caractères indiqués. Avant que les conversions ne commencent, la plupart d'entre elles éliminent les blancs. Ces espaces blancs ne sont pas comptés dans le champ de largeur maximale.

Les conversions suivantes sont disponibles :

% Correspond à un caractère %. Ceci signifie qu'un indicateur %% dans la chaîne de format correspond à un seul caractere % dans la chaine d'entrée. Aucune conversion, et aucune assignation n'a lieu.

d Correspond à un entier décimal éventuellement signé, le pointeur correspondant doit être du type `int *`. D Equivalent à ld, utilisé uniquement pour compatibilité avec des versions précédentes. (Et seulement dans libc4. Dans libc5 et glibc le %D est ignoré silencieusement, ce qui conduit d'anciens programmes à échouer mystérieusement). i correspond à un entier éventuellement signé. Le pointeur suivant doit être du type `int *`. L'entier est en base 16 (hexadécimal) s'il commence par 0x ou 0X, en base 8 (octal) s'il commence par un 0, et en base 10 sinon. Seuls les caractères correspondants à la base concernée sont utilisés.

o Correspond à un entier octal non signé. Le pointeur correspondant doit être du type `unsigned int *`.

u Correspond à un entier décimal non signé. Le pointeur suivant doit être du type `unsigned int *`.

x Correspond à un entier héxadécimal non signé. Le pointeur suivant doit être du type `unsigned int *`.

X Equivalent à x

f Correspond à un nombre réel éventuellement signé. Le pointeur correspondant doit être du type `float *`.

e Equivalent à f.

g Equivalent à f.

E Equivalent à f.

s Correspond à une séquence de caractères différents des caractères blancs. Le pointeur correspondant doit être du type `char *`, et la chaine doit être assez large pour accueillir toute la séquence, ainsi que le caractère NUL final. La conversion s'arrête au premier caractère blanc, ou à la longueur maximale du champ.

c Correspond à une séquence de width caractères (par défaut 1). Le pointeur associé doit être du type `char *`, et il doit y avoir suffisament de place dans la chaîne pour tous les caracteres. Aucun caractère NUL final n'est ajouté. Les caractères blancs de début ne sont pas supprimés. Si on veut les éliminer, il faut utiliser un espace dans le format.

[Correspond à une séquence non vide de caractères appartenant à un ensemble donné. Le pointeur correspondant doit être du type `char *`, et il doit y avoir suffisament de place dans le tableau de caractères pour accueillir la chaîne ainsi qu'un caractère NUL final. Les caractères blancs du début ne sont pas supprimés. La chaîne est constituée de caractères inclus ou exclus d'un ensemble donné. L'ensemble est composé des caractères compris entre les deux crochets [et]. L'ensemble exclut ces caractères si le premier après le crochet ouvrant est un accent circonflexe ^. Pour inclure un crochet fermant dans l'ensemble, il suffit de le placer en première position après le crochet ouvrant, ou l'accent circonflexe ; à tout autre emplacement il servira à terminer l'ensemble. Le caractère tiret - a également une signification particulière. Quand il est placé entre deux autres caractères, il ajoute à l'ensemble les caractères intermédiaires. Pour inclure un tiret dans l'ensemble, il faut le placer en dernière position avant le crochet fermant. Par exemple, `[^]0-9-]` correspond à l'ensemble « Tout sauf le crochet fermant, les chiffres de 0 à 9, et le tiret ». La chaine se termine dès l'occurence d'un caractère exclu (ou inclus s'il y a un accent circonflexe) de l'ensemble, ou dès qu'on atteint la longueur maximale du champ.

p Correspond à une valeur de pointeur (comme affichée par `%p` dans printf(3)). Le pointeur correspondant doit être du type `void *`.

n Aucune lecture n'est faite. Le nombre de caractères déjà lus est stocké dans le pointeur correspondant, qui doit être de type `int *`. Ce n'est pas une conversion, mais le stockage peut quand même être supprimé avec un attribut *. Le standard C indique : « L'exécution d'une directive `%n` n'incrémente pas le compteur d'assignations renvoyé à la fin de l'exécution ». Mais il semble qu'il y ait des contradictions sur ce point. Il est probablement sage de ne pas faire de suppositions sur l'effet de la conversion `%n` sur la valeur renvoyée.

fseek, rewind — POSITIONNEMENT DANS UN FICHIER — fseek, rewind

Prototype
```
#include <stdio.h>
int fseek (FILE *stream, long offset, int whence);
void rewind (FILE *stream);
```

Description La fonction fseek fixe l'indicateur de position du flux pointé par stream. La nouvelle position, mesurée en octets, est obtenue en additionnant offset octets au point de départ indiqué par whence. Si whence vaut SEEK_SET, SEEK_CUR, ou SEEK_END, le point de départ correspond respectivement au début du fichier, à la position actuelle, ou à la fin du fichier. Un appel réussi à fseek efface l'indicateur de fin de fichier pour le flux, et annule les effets de toute fonction ungetc(3) précédente sur le même flux.

La fonction rewind ramène l'indicateur de position du flux pointé par stream au début du fichier. C'est l'équivalent de :

```
(void)fseek(stream, 0L, SEEK_SET)
```

sauf que l'indicateur d'erreur du flux est également effacé.

Valeur renvoyée La fonction rewind ne renvoie pas de valeur. Si elle réussit totalement, fseek renvoie 0. Sinon, elle renvoie -1 et la variable globale errno contient le code d'erreur.

feof, ferror, clearerr — ETATS DE FLUX — feof, ferror, clearerr

Prototype
```
#include <stdio.h>
int feof (FILE *stream);
int ferror (FILE *stream);
void clearerr (FILE *stream);
```

Description La fonction feof teste l'indicateur de fin de fichier concernant le flux pointé par stream, et renvoie une valeur non nulle si cet indicateur est actif. L'indicateur de fin de fichier ne peut être réinitialisé que par la fonction clearerr.

La fonction ferror teste l'indicateur d'erreur concernant le flux pointé par stream, et envoie une valeur non nulle si cet indicateur est actif. L'indicateur d'erreur ne peut être réinitialisé que par la fonction clearerr.

La fonction clearerr efface les indicateurs d'erreur et de fin de fichier concernant le flux pointé par stream.

B.2 Gestion de la mémoire

La gestion de la mémoire se fait par l'intermédiaire de la bibliothèque <stdlib.h>.

---

malloc, calloc, free, realloc — ALLOCATION DYNAMIQUE — malloc, calloc, free, realloc

---

Prototype
```
#include <stdlib.h>
void *calloc (size_t nmemb, size_t size);
void *malloc (size_t size);
void free (void *ptr);
void *realloc (void *ptr, size_t size);
```

Description calloc() alloue la mémoire nécessaire pour un tableau nmemb éléments, chacun d'eux représentant size octets, et renvoie un pointeur vers la mémoire allouée. Cette zone est remplie avec des zéros.

malloc() alloue size octets, et renvoie un pointeur sur la mémoire allouée. Le contenu de la zone de mémoire n'est pas initialisé.

free() libère l'espace mémoire pointé par ptr, qui a été obtenu lors d'un appel antérieur à malloc(), calloc() ou realloc(). Si le pointeur ptr n'a pas été obtenu par l'un de ces appels, ou si il a déjà été libéré avec free(), le comportement est indéterminé. Si ptr est NULL, aucune tentative de libération n'a lieu.

realloc() modifie la taille du bloc de mémoire pointé par ptr pour l'amener à une taille de size octets. realloc() conserve le contenu de la zone mémoire minimum entre la nouvelle et l'ancienne taille. Le contenu de la zone de mémoire nouvellement allouée n'est pas initialisé. Si ptr est NULL, l'appel de realloc() est équivalent à malloc(size). Si size vaut zéro, l'appel est équivalent à free(ptr) . Si ptr n'est pas NULL, il doit avoir été obtenu par un appel antérieur à malloc(), calloc() ou realloc().

Valeur renvoyée Pour calloc() et malloc(), la valeur renvoyée est un pointeur sur la mémoire allouée, qui est correctement alignée pour n'importe quel type de variable, ou NULL si la demande échoue.

free() ne renvoie pas de valeur.

realloc() renvoie un pointeur sur la mémoire nouvellement allouée, qui est correctement alignée pour n'importe quel type de variable, et qui peut être différent de ptr, ou NULL si la demande échoue, ou si size vaut zéro. Si realloc() échoue, le bloc mémoire original reste intact, il n'est ni libéré ni déplacé.

B.3 Fonctions diverses, conversions, tests

---

isalpha,... — FONCTIONS SUR CARACTÈRES — isalpha,...

Prototype
```
#include <ctype.h>
int isalnum (int c);
int isalpha (int c);
int isascii (int c);
int isblank (int c);
int iscntrl (int c);
int isdigit (int c);
```

```
int isgraph (int c);
int islower (int c);
int isprint (int c);
int ispunct (int c);
int isspace (int c);
int isupper (int c);
int isxdigit (int c);
```

Description Ces fonctions vérifient si le caractère c, qui doit avoir la valeur d'un unsigned char ou valoir EOF, rentre dans une catégorie donnée, en accord avec la localisation en cours.

isalnum() vérifie si l'on a un caractère alphanumérique. C'est équivalent à (isalpha(c) || isdigit(c)).

isalpha() vérifie si l'on a un caractère alphabétique. Dans la localisation "C" standard, c'est équivalent à (isupper(c) || islower(c)). Dans certaines localisations, il peut y avoir des caractères supplémentaires pour lesquels isalpha() est vrai–des lettres qui ne sont ni majuscules ni minuscules.

isascii() vérifie si c est un unsigned char sur 7 bits, entrant dans le jeu de caractères ASCII. Cette fonction est une extension BSD et SVID.

isblank() vérifie si le caractère est blanc, c'est à dire un espace ou une tabulation. C'est une extension GNU.

iscntrl() vérifie si l'on a un caractère de contrôle.

isdigit() vérifie si l'on a un chiffre (0 a 9).

isgraph() vérifie s'il s'agit d'un caractère imprimable, à l'exception de l'espace.

islower() vérifie si l'on a un caractère minuscule.

isprint() vérifie s'il s'agit d'un caractère imprimable, y compris l'espace.

ispunct() vérifie s'il s'agit d'un caractère imprimable, qui ne soit ni un espace, ni un caractere alphanumérique.

isspace() vérifie si l'on a un caractère blanc, d'espacement. Dans les localisations "C" et "POSIX" il s'agit de : espace, saut de page (form-feed), saut de ligne (newline), retour chariot (carriage return), tabulation horizontale, et tabulation verticale.

isupper() vérifie si l'on a une lettre majuscule.

isxdigit() vérifie s'il s'agit d'un chiffre héxadécimal, c'est à dire 0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F.

Valeur renvoyée Les valeurs renvoyées sont non nulles si le caractère c entre dans la catégorie testée, et zéro sinon.

---

strcmp, strncmp — Comparaison de chaînes — strcmp, strncmp

Prototype `#include <string.h>`

```
int strcmp (const char *s1, const char *s2);
int strncmp (const char *s1, const char *s2, size_t n);
```

Description La fonction strcmp() compare les deux chaînes s1 et s2. Elle renvoie un entier négati, nul, ou positif, si s1 est respectivement inférieure, égale ou supérieure à s2.

La fonction strncmp() est identique sauf qu'elle ne compare que les n premiers caractères de s1.

Valeur renvoyée Les fonctions strcmp() et strncmp() renvoient un entier inférieur, égal ou supérieur à zéro si s1 (ou ses n premiers octets) est respectivement inférieure, égale ou supérieure à s2.

---

strcat, strncat — Concaténation de chaînes — strcat, strncat

Prototype
```
#include <string.h>
char *strcat (char *dest, const char *src);
char *strncat (char *dest, const char *src, size_t n);
```

Description La fonction strcat() ajoute la chaîne src à la fin de la chaîne dest en écrasant le caractère '\0' à la fin de dest, puis en ajoutant un nouveau caractère '\0' final. Les chaînes ne doivent pas se chevaucher, et la chaîne dest doit être assez grande pour accueillir le résultat.

La fonction strncat() est identique, à la différence qu'elle ne copie que les n premiers caractères de src a la fin de dest.

Valeur renvoyée Les fonctions strcat() et strncat() renvoient un pointeur sur la chaîne résultat dest.

---

strcpy, strncpy — Copie de chaîne — strcpy, strncpy

Prototype
```
#include <string.h>
char *strcpy (char *dest, const char *src);
char *strncpy (char *dest, const char *src, size_t n);
```

Description La fonction strcpy() copie la chaîne pointée par src (y compris le caractère '\0' final) dans la chaîne pointée par dest. Les deux chaînes ne doivent pas se chevaucher. La chaîne dest doit être assez grande pour accueillir la copie.

La fonction strncpy() est identique, sauf que seuls les n premiers octets de src sont copiés. Ainsi, s'il n'y a pas de caractère nul dans les n premiers octets de src, la chaîne résultante ne disposera de caractère nul final.

Dans le cas où la longueur src est inférieure à n, la fin de dest sera remplie avec des caractères nuls.

Valeur renvoyée Les fonctions strcpy() et strncpy() renvoient un pointeur sur la chaîne destination dest.

---

strlen — Longueur d'une chaîne — strlen

Prototype
```
#include <string.h>
size_t strlen (const char *s);
```

Description La fonction strlen() calcule la longueur de la chaîne de caractères s, sans compter le caractere nul '\0' final.

Valeur renvoyée La fonction strlen() renvoie le nombre de caractères dans la chaîne s.

---

strchr — Recherche d'un caractère dans une chaîne — strchr

Prototype
```
#include <string.h>
char *strchr (const char *s, int c);
char *strrchr (const char *s, int c);
```

Description La fonction strchr() renvoie un pointeur sur la première occurence du caractère c dans la chaîne s.

La fonction strrchr() renvoie un pointeur sur la dernière occurence du caractère c dans la chaîne s.

Valeur renvoyée Les fonctions strchr() et strrchr() renvoient un pointeur sur le caractère correspondant, ou NULL si le caractère n'a pas été trouvé.

---

strstr — Recherche d'une sous-chaîne — strstr

Prototype
```
#include <string.h>
char *strstr (const char *meule_de_foin, const char *aiguille);
```

Description La fonction strstr() cherche la premiere occurence de la sous-chaîne aiguille au sein de la chaîne meule_de_foin. Les caractères '\0' de fin ne sont pas comparés.

Valeur renvoyée La fonction strstr() renvoie un pointeur sur le début de la sous-chaîne, ou NULL si celle-ci n'est pas trouvée.

---

atof — Conversion en flottant — atof

Prototype
```
#include <stdlib.h>
double atof (const char *nptr);
```

Description La fonction atof() convertit le début de la chaîne pointée par nptr en un réel de type double. Le résultat est identique à un appel

```
strtod(nptr, (char **)NULL);
```

à la différence que atof() ne détecte pas d'erreur.

Valeur renvoyée Le résultat de la conversion.

---

atoi — Conversion en entier — atoi

Prototype
```
#include <stdlib.h>
int atoi (const char *nptr);
```

Description La fonction atoi() convertit le début de la chaîne pointée par nptr en entier de type int. Le résultat est identique à un appel :

```
strtol(nptr, (char **)NULL, 10);
```

à la différence que atoi() ne détecte pas d'erreur.

Valeur renvoyée Le résultat de la conversion.

---

atol — Conversion d'un entier en long — atol

Prototype
```
#include <stdlib.h>
long atol (const char *nptr);
```

Description La fonction atol() convertit le début de la chaîne pointée par nptr en un entier de type long. Le résultat est identique à un appel :

```
strtol(nptr, (char **)NULL, 10);
```

à la différence que atol() ne détecte pas d'erreur.

Valeur renvoyée Le résultat de la conversion.

B.4 Mathématiques

---

fabs, abs — Valeur absolue — fabs, abs

Prototype
```
#include <math.h>
double fabs (double x);

#include <stdlib.h>
int abs (int j);
```

Description La fonction fabs() renvoie la valeur absolue du nombre réel x.

La fonction abs() calcule la valeur absolue de l'argument entier j.

---

floor, ceil — Calcul d'arrondis — floor, ceil

Prototype
```
#include <math.h>
double floor (double x);
double ceil (double x);
```

Description La fonction floor() renvoie la valeur x arrondie par défaut, sous forme de réel (double).

La fonction ceil() arrondit x par excès à l'entier le plus proche, qui est renvoyé sous forme de réel (double).

---

exp, log, log10, pow — Exponentielle, log et puissance — exp, log, log10, pow

---

Prototype
```
#include <math.h>
double exp (double x);
double log (double x);
double log10 (double x);
double pow (double x, double y);
```

Description La fonction exp() renvoie la valeur de e (la base des logarithmes naturels), élevée à la puissance x.

La fonction log() renvoie le logarithme naturel (ou logarithme néperien) de x, noté traditionnellement ln.

La fonction log10() renvoie le logarithme décimal de x.

La fonction pow() renvoie la valeur de x élevé à la puissance y.

sin, cos,... FONCTIONS TRIGONOMÉTRIQUES sin, cos,...

Prototype
```
#include <math.h>
double sin (double x);
double cos (double x);
double tan (double x);
double asin (double x);
double acos (double x);
double atan (double x);
```

Description La fonction sin() renvoie le Sinus du réel x, ce dernier étant fourni en radians. La fonction cos() renvoie le Cosinus du réel x, ce dernier étant fourni en radians. La fonction tan() renvoie la tangente de x, ce dernier étant fourni en radians.

La fonction asin() calcule l'Arc Sinus de x, c'est à dire la valeur dont le sinus est x. Si x se trouve en dehors de l'intervalle [-1, 1], asin() reverra NaN (Not a Number), et errno contiendra le code d'erreur EDOM.

La fonction acos() calcule l'Arc Cosinus de x, c'est à dire la valeur dont le cosinus est x. Si x se trouve en dehors de l'intervalle [-1, 1], acos() renverra NaN (Not a Number), et errno contiendra le code d'erreur EDOM.

La fonction atan() calcule l'Arc Tangente de x, c'est à dire la valeur dont la tangente est x. Le domaine de définition de cette fonction étant l'ensemble des réels, elle ne renvoie jamais d'erreur.

Valeur renvoyée La fonction sin() renvoie une valeur réelle entre -1 et 1. La fonction cos() renvoie une valeur réelle entre -1 et 1.

La fonction asin() renvoie l'Arc Sinus en radians, dans l'intervalle -PI/2 à PI/2 (bornes comprises). La fonction acos() renvoie l'Arc Cosinus en radians, dans l'intervalle 0 a PI (0 et PI compris). La fonction atan() renvoie l'Arc Tangente en radians dans l'intervalle [-PI/2, PI/2] (bornes comprises).

B.5 Autres fonctions

rand, srand GÉNÉRATEUR DE NOMBRES ALÉATOIRES rand, srand

Prototype
```
#include <stdlib.h>
int rand (void);
void srand (unsigned int seed);
```

Description La fonction rand() renvoie un entier pseudo-aléatoire entre 0 et RAND_MAX.

La fonction srand() utilise son argument comme "graine" pour la génération d'une nouvelle séquence de nombres pseudo-aléatoires, qui seront fournis par rand(). Ces séquences sont reproductibles en appelant srand() avec la même valeur de graine.

Si aucune graine originale n'est fournie, la fonction rand() commence en utilisant la valeur 1.

Valeur renvoyée La fonction rand() renvoie un nombre entier entre 0 et RAND_MAX. La fonction srand() ne renvoie aucune valeur.

time — Obtenir la date et l'heure — time

Prototype
```
#include <time.h>
time_t time(time_t *t);
```

Description time renvoie l'heure actuelle sous forme du nombre de secondes écoulées depuis le 1er Janvier 1970 à 00h 00m 00s GMT, le début de l'Epoque (Epoch en anglais).

Si t n'est pas NULL, la valeur renvoyée est également stockée dans la structure vers laquelle il pointe.

Valeur renvoyée S'il réussit, l'appel time renvoie l'heure actuelle. S'il échoue, la valeur ((time_t)-1) est renvoyée, et errno contient le code d'erreur.

---

ctime,... — Formater une date/heure — ctime,...

Prototype
```
#include <time.h>
char *asctime (const struct tm *timeptr);
char *ctime (const time_t *timep);
struct tm *gmtime (const time_t *timep);
struct tm *localtime (const time_t *timep);
```

Description Les fonctions ctime(), gmtime() et localtime() prennent toutes un argument de type time_t qui représente une date. Si l'on interprète cet argument comme une valeur absolue, il s'agit du nombre de secondes écoulées depuis le 1er Janvier 1970 à 00h 00m 00s en Temps Universel (TU). On peut obtenir cette valeur grâce à la fonction time().

La fonction ctime() convertit la date timep en une chaîne de caractères de la forme :

```
"Wed Jun 30 21:49:08 1993\n"
```

la fonction gmtime() convertit la date timep en une représentation struct tm exprimée en Temps Universel.

La fonction localtime() convertit la date timep en une représentation struct tm exprimée en fonction du fuseau horaire de l'utilisateur.

La fonction asctime() convertit la date timeptr exprimée sous forme struct tm en une chaîne de caractères du même format que ctime(). La valeur renvoyée pointe sur une chaîne statique qui sera écrasée à chaque appel de l'une des fonctions ci-dessus.

INDEX

Éric Guérin
eric.guerin@liris.cnrs.fr