APPRENTISSAGE DU LANGAGE "C"

Licence de ce document

Registre des éditions

Version	Date	Description des modifications	Auteur des modifications
1.0	2000	Création du document	Frédéric Lang (fr.lang@free.fr)
1.1	6 février 2004	Modification de l'exercice "Triangle de Pascal" Refonte du chapitre sur les pointeurs Ajout du registre des éditions Ajout de la licence de ce document Ajout d'exemples et exercices	Frédéric Lang (fr.lang@free.fr)
1.2	20 février 2004	Corrections mineures dans l'exercice "Triangle de Pascal" Ajout de la fonction "strerror()" Corrections mineures (fautes d'othographes, etc.)	Frédéric Lang (fr.lang@free.fr)
1.3	27 avril 2004	Ajout des fonctions "truncate()" et "ftruncate()" Modification des en-têtes	Frédéric Lang (fr.lang@free.fr)
1.4	30 juillet 2004	Modification de petits détails sur les structures et sur les options de compilation	Frédéric Lang (fr.lang@free.fr)
1.5	26 août 2004	Ajout des types "long long" et "long double" Ajout des fonctions "getline()" et "getdelim()" Ajout des fonctions "fpurge()" et "__fpurge()" Modifications de mise en page	Frédéric Lang (fr.lang@free.fr)
1.6	16 mars 2005	Rajout des notions "stdin", "stdout" et "stderr" Rajout des fonctions "getc()" et "putc()" Rajout des fonctions "getchar()" et "putchar()" Rajout des fonctions "gets()" et "puts()" Correction de la fonction "fflush" Corrections mineures	Frédéric Lang (fr.lang@free.fr)
2.0	5 avril 2005	Modification du titre du document Rajout de la notation constante en "float" Rajout du type des fonctions "main" dans les exemples et exercices	Frédéric Lang (fr.lang@free.fr)

SOMMAIRE

I) GENERALITES

La programmation en C, c'est comme le sexe chez les adolescents
Tout le monde y pense
Tout le monde en parle
Tout le monde croit que le voisin le fait
Presque personne ne le fait
Ceux qui le font le font mal
Pensent que la prochaine fois ce sera mieux
Ne prennent pas de précautions
N'osent pas avouer leurs lacunes de peur de paraître niais
Sont fort bruyants quand ils y arrivent

1) Introduction – Historique

Le langage C est un langage de programmation conçu pour de multiples utilisations. Son développement est parallèle au système UNIX car le noyau du système UNIX est écrit en langage C. Désormais, il est utilisé sur tous les systèmes d'exploitation et donc sur toutes les machines.

Les principes fondamentaux du langage C sont issus des langages B et BCPL créés vers 1970 pour le premier système UNIX dans les laboratoires AT&T par Ken Thompson et Dennis Ritchie.
En 1970, Ken Thompson crée le langage B inspiré du BCPL.
En 1972, Dennis Ritchie définit le langage C à partir des travaux de Ken Thompson.

Ces travaux seront continués par Brian W Kernigham et Dennis Ritchie en 1972 pour donner naissance au premier ouvrage de référence "Le langage C" éditions MASSON.

Si le noyau du langage C est assez restreint, il s'enrichit d'une librairie standard qui contient des fonctions permettant les traitements les plus divers (Entrées-Sorties, Fichiers, Traitements des chaînes de caractères, Gestion dynamique de la mémoire, etc.).

L'étude du système UNIX passe obligatoirement par l'étude du langage C. Ultérieurement, il sera possible d'aborder l'étude de la POO (Programmation Orientée Objet) en C++.

Des travaux de normalisation ont été entrepris à partir de 1983. Le C-ANSI a été approuvé fin 1988 (Norme X3-159-1989).

Dans ce document, on s'efforcera d'utiliser la norme ANSI. Tous les exemples (sauf avertissement) seront donnés pour un compilateur UNIX (*cc* ou *gcc*) respectant la norme ANSI.

2) Notions de fonctions et de blocs

Les unités de structurations du langage C sont :

- la fonction qui correspond à la notion de sous-programme en PASCAL, FORTRAN ou ADA. Il n'y a pas de distinction entre procédure (qui effectue une action mais qui ne renvoie rien) et fonction (qui calcule et renvoie un résultat en fonction de paramètres).
- les fichiers comprenants des déclarations de fonctions et d'objets. Les fichiers peuvent être compilés séparéments. Ils constituent des modules, briques de base dans la construction d'applications.

Un programme C, c'est en fait un ou plusieurs fichiers comprenant :

- des déclarations de fonctions.
- des déclarations d'objets, variables ou constantes.
- des instructions de travail
- des directives pour le pré processeur.

Contrairement à PASCAL ou ADA, on ne peut déclarer de fonction à l'intérieur d'une autre fonction.

Une fonction est un module (ou sous programme) destinée à effectuer une opération élémentaire et une seule, en définissant les cas limites. En langage C, un programme source est constitué de fonctions dont une particulière (appelée **main**) constitue le point d'entrée et de sortie du programme. On doit donc impérativement trouver "**main**" dans un programme et on ne doit en trouver qu'un seul.

Un bloc est un ensemble d'instructions formant une entité logique indissociable bornée par une accolade ouvrante "**{**" (début) et une accolade fermante "**}**" (fin).Le plus grand bloc est la fonction… mais celle-ci peut contenir d'autres blocs d'instructions.

Exemple de programme en C :

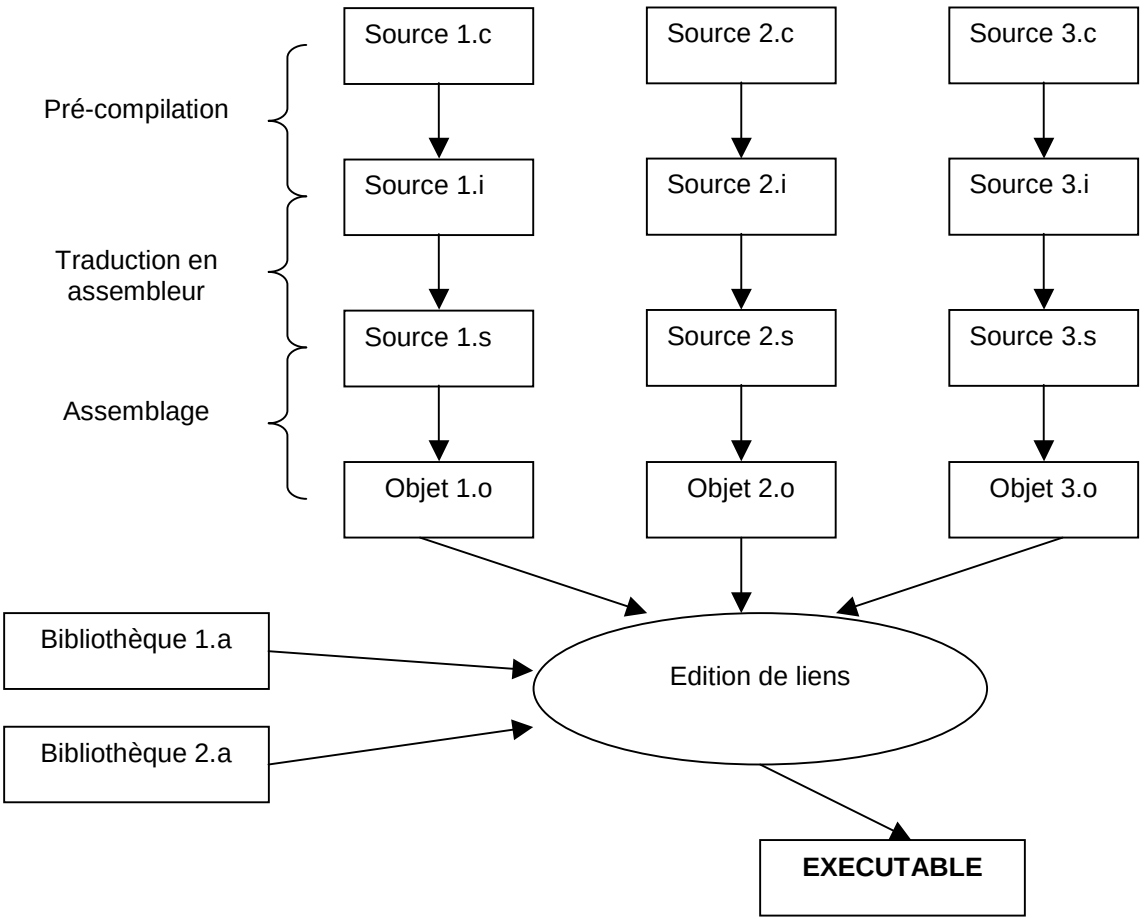
```
fonction1(…)                    // Fonction 1 – Point d'entrée dans la fonction 1
{
        Variables éventuelles
        /* Commentaires éventuels /
        Instructions
}                               // Point de sortie de la fonction 1

main(…)                         // Fonction principale - Point d'entrée du programme
{
        Variables éventuelles
        /* Commentaires /
        Instructions
}                               // Point de sortie du programme

fonctionN(…)                    // Fonction N – Point d'entrée dans la fonction N
{
        Variables éventuelles
        /* Commentaires éventuels /
        Instructions
}                               // Point de sortie de la fonction N
```

3) Le compilateur C sous "Unix"

L'ensemble d'un exécutable peut tenir dans plusieurs fichiers sources ; chacun contenant une ou plusieurs fonctions utilisées par le programme. La compilation d'un programme consistera alors à transformer chaque source en un module "objet" et à relier ensuite les différents modules "objet" pour produire un exécutable.



Le passage du fichier source au module "objet" est généralement automatique pour le compilateur (sauf si on lui demande expressément de s'arrêter en chemin). De plus, comme il existe très souvent qu'un seul source par exécutable, le compilateur continue généralement son travail jusqu'à produire l'exécutable.
La commande de compilation sous UNIX est :

cc [options] liste de fichiers

Extension des fichiers

.c	Fichier source
.i	Fichier interprétable par le compilateur
.s	Module assembleur
.o	Fichier objet
.a	Librairie statique
sans extension	Exécutable

Quelques exemples de la commande "*cc*"

cc fic.c	→	Compilation de "**fic.c**". L'exécutable généré est automatiquement nommé "**a.out**"
cc fic.c -o prog	→	Compilation de "**fic.c**". L'exécutable généré est nommé "**prog**"
cc -O fic.c -o fic	→	Compilation optimisée de "**fic1.c**". L'exécutable généré est nommé "**fic**"
cc -c fic.c	→	Compilation partielle de "**fic.c**". La compilation s'arrête au module objet nommé "**fic.o**"
cc -S fic.c	→	Compilation partielle de "**fic.c**". La compilation s'arrête à la création d'un source en assembleur nommé "**fic.s**"
cc -P fic.c	→	Compilation partielle de "**fic.c**". La compilation s'arrête à la création d'un source "**fic.i**" directement compilable.
cc fic1.c fic2.c -o prog	→	Compilation des deux sources liés "**fic1.c**" et "**fic2.c**" pour produire un exécutable "**prog**"
cc fic1.o fic2.o -o prog	→	Compilation des deux modules objets liés "**fic1.o**" et "**fic2.o**" pour produire un exécutable "**prog**"
cc fic.c libtruc.a -o prog	→	Compilation de "**fic.c**" en le liant à la librairie "**libtruc.a**" pour produire un exécutable "**prog**"
cc fic.c –lm -o prog	→	Compilation de "**fic.c**" en le liant à la librairie mathématique "**/usr/lib/libm.a**" pour produire un exécutable "**prog**". L'option "**-l<qqchose>**" est un raccourci vers "**/usr/lib/libqqchose.a**"
cc -g fic.c	→	Compilation de "**fic.c**" avec insertion automatique d'instructions de débogage L'exécutable généré est automatiquement nommé "**a.out**"
cc fic.c -qsrcmsg	→	Compilation de "**fic.c**" avec affichage de toutes les lignes contenant des erreurs si erreur il y a. L'exécutable généré est automatiquement nommé "**a.out**"
cc fic.c -qxref	→	Compilation de "**fic.c**" et création d'un listing "**fic.lst**" qui donne les références croisées (quelle fonction appelle quelle autre fonction) L'exécutable généré est automatiquement nommé "**a.out**"

Remarques : la librairie standard "**/usr/lib/libc.a**" contenant une multitude de fonctions de travail est toujours automatiquement liée lors de la compilation.

4) La documentation C sous "Unix"

Chaque fonction du langage est expliquée dans la documentation intégrée d'Unix (commande "***man***"). En cas de besoin, l'accès est toujours possible.

Exemple : Pour une aide sur la fonction "printf"

```
man printf
```

II) ELEMENTS DE BASE DU LANGAGE

1) Les identificateurs

Un identificateur permet de nommer une variable, une pseudo constante (sera vue en fin de cours) ou une fonction. Ils doivent être écrits selon les règles suivantes :

- 32 caractères maximum (certains systèmes ne tiennent compte que des 8 premiers).
- le premier caractère doit obligatoirement être soit une lettre soit le caractère "souligné" (mais c'est à éviter pour la clarté du source).
- les autres caractères peuvent être indifféremment des lettres (a-z), des chiffres (0-9) ou le caractère "souligné".
- le signe "moins", l'espace, parfois les caractères accentués ainsi que tous les caractères non cités précédemment sont interdits.

En règle générale, le nom des variables est écrit en lettres minuscules avec quelques majuscules (pour faire ressortir le but de la variable) et celui des pseudo constantes (*cf. chapitre sur le préprocesseur)* totalement en majuscules (conventions des programmeurs).

Le compilateur C fait la différence entre majuscules et minuscules.

Certains identificateurs sont interdits car ils constituent les mots réservés du langage (vu plus loins).

Autorisés	Interdits	Raison de l'interdiction
i	Cpt-Lig	Présence du caractère "**moins**"
j	a z	Présence de l'**espace**
CompteurLignes	3f25	Commence par un **chiffre**
Compteur2_Col	$a	Présence du caractère "**dollar**"

2) Les mots réservés

Certains mots sont réservés par le langage car ils ont une signification particulière. Il est alors interdit d'utiliser un de ces mots comme identificateur. Bien que le compilateur fasse la différence entre majuscules et minuscules et qu'on puisse donc utiliser un de ces mots en majuscules comme identificateur, il est préférable, pour la clarté du source, d'éviter cette attitude de programmation.

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

3) Les séparateurs

Les différents séparateurs reconnus par le compilateur peuvent avoir plusieurs significations.

Type	Nom	Significations
[…]	Crochets	Indice d'accès dans un tableau
(…)	Parenthèses	1/ Groupement d'expressions (force la priorité) 2/ Isolement des expressions conditionnelles 3/ Déclaration des paramètres d'une fonction 4/ Conversion explicite d'un type (casting)
{…}	Accolades	1/ Début et fin de bloc d'instructions 2/ Initialisation à la déclaration d'un tableau 3/ Déclaration d'une structure
,	Virgule	1/ Sépare les éléments d'une liste d'arguments 2/ Concaténation d'instructions
;	Point virgule	Terminateur d'instruction
:	Deux points	Label
.	Point	Accède à un champ d'une structure
->	Flèche ("moins" suivi de "supérieur")	Accède à un champ d'un pointeur sur une structure

4) Les commentaires

Les commentaires permettent de porter des remarques afin de faciliter la lecture d'un programme source. Chaque commentaire est délimité par les combinaisons suivantes "***/* Commentaire */***"
Les commentaires ne sont pas pris en compte par le compilateur et n'augmentent donc pas la taille des programmes exécutables.

Les commentaires peuvent tenir sur plusieurs lignes du style :

```
/*      Je commente mon programme
        Je continue mes commentaires
        J'ai fini mes commentaires      */
```

Les commentaires ne peuvent pas être imbriqués comme dans le style :

```
/* Début Commentaire 1 /* Commentaire 2 */ Fin commentaire 1*/
```

Remarque : Depuis l'avènement du C^{++}, les compilateurs acceptent maintenant comme commentaire la séquence "**//**" spécifique à ce langage ; qui permet de ne commenter qu'une ligne ou une partie de la ligne.

```
// Cette ligne est totalement mise en commentaire
int i;                  // Le commentaire ne commence qu'à partie de la séquence "//"
```

5) Les variables

Comme tout langage déclaratif, le langage C exige que les variables soient déclarées avant leur utilisation. En fait, pour chaque bloc d'instructions délimité par des accolades "**{**", il faut déclarer toutes les variables avant que soit écrite la première instruction du bloc. Mais cette règle étant valable pour chaque bloc ; on peut aussi déclarer des variables dans des sous-bloc d'instructions.
Exemple :

```
// Bloc d'instruction n°1
{
	Déclaration des variables (avant toute instruction du bloc n°1)
	Instruction
	Instruction
	…
	// Sous-bloc d'instruction n°2
	{
		Déclaration des variables (avant toute instruction du bloc n°2)
		Instruction
		Instruction
		…
	}
	…
	Instruction appartenant au bloc n°1
}
```

Une variable est définie par plusieurs attributs; certains facultatifs mais ayants alors une valeur par défaut) :

- sa classe d'allocation en mémoire ("auto" par défaut)
- en cas de variable numérique, l'indication "signé" ou "non-signé" (signé par défaut)
- le type de la valeur qu'elle doit stocker (entier par défaut, nombre en virgule flottante, etc.)
- son nom (l'identificateur de la variable)

Déclaration :
[classe] [unsigned/signed] <type> <identificateur>; (n'oubliez pas le point-virgule final)

Exemple :

```
static int nb;	// Déclaration d'une variable nb de type int de classe static
int i;		// Déclaration d'une variable i de type int (de classe auto par défaut)
```

Remarque : la notion de classe sera revue plus en détail par la suite du cours. Jusque là, il suffira de ne pas spécifier de classe pour vos variables.

Lors de sa création en mémoire, une variable possède n'importe quelle valeur.
Il convient donc de ne jamais présumer de la valeur d'une variable non-initialisée !!!

III) ÉLEMENTS DE BASE POUR DEBUTER

1) Les types de base

Les types de base correspondent aux types directement supportés par la machine. Dans le langage C, tout type de base est un nombre codé sur un ou plusieurs octets. Donc tout type de base accepte toute opération mathématique de base !!!

a) Les entiers

- **int :** entier codé sur 2 ou 4 octets suivant la machine sur laquelle on travaille. Dans le monde Unix, il est généralement de 4 octets. Sa plage de valeur peut donc aller de :
  - -2^{31} à $2^{31} - 1$; c'est à dire de –2 147 483 648 à 2 147 483 647 s'il est déclaré signé
  - 0 à $2^{32} - 1$; c'est à dire de 0 à 4 294 967 296 s'il est déclaré non-signé
- **short int** : entier court codé sur 2 octets. Le mot "**short**" est suffisant par lui-même donc le mot "**int**" n'est pas obligatoire. Sa plage de valeur peut aller de :
  - -2^{15} à $2^{15} - 1$; c'est à dire de –32 768 à 32 767 s'il est déclaré signé
  - 0 à $2^{16} - 1$; c'est à dire de 0 à 65 535 s'il est déclaré non-signé
- **long int** : entier long codé sur 4 octets. Le mot "**long**" est suffisant par lui-même donc le mot "**int**" n'est pas obligatoire. Sa plage de valeur peut aller de :
- -2^{31} à $2^{31} - 1$; c'est à dire de –2 147 483 648 à 2 147 483 647 s'il est déclaré signé
  - 0 à $2^{32} - 1$; c'est à dire de 0 à 4 294 967 296 s'il est déclaré non-signé
- **long long int** : entier long codé sur 8 octets. Les mots "**long long**" sont suffisants par eux-mêmes donc le mot "**int**" n'est pas obligatoire. Sa plage de valeur peut aller de :
- -2^{63} à $2^{63} - 1$; c'est à dire de – 9 223 372 036 854 775 808 à 9 223 372 036 854 775 807 s'il est déclaré signé
  - 0 à $2^{64} - 1$; c'est à dire de 0 à 18 446 744 073 709 551 615 s'il est déclaré non-signé
- **char** : caractère ASCII qui est un nombre entier codé sur 1 octet. Lui aussi de par sa nature numérique accepte des opérations mathématiques. Sa plage de valeur peut aller de :
  - -2^{7} à $2^{7} - 1$; c'est à dire de –128 à 127 s'il est déclaré signé
  - 0 à $2^{8} - 1$; c'est à dire de 0 à 255 s'il est déclaré non-signé

Remarques :

- Ajouter 1 à une variable ayant atteint sa limite maximale la fait basculer en limite minimale (perte de la retenue). C'est à dire que "127 + 1 = -128" si on travaille sur une variable de type "char signé"
- Il est possible de forcer un entier à être signé ou non-signé en rajoutant le mot clef "***signed***" ou "***unsigned***" avant son type (char, int, short int, long int). Dans le monde Unix, une variable est signée par défaut donc le mot clef "***signed***" est inutile.

b) Les réels

Ils respectent les règles de codage IBM des réels utilisant un bit pour le signe, "x" bits pour la mantisse et "y" pour l'exposant. Ils sont de deux types standard et un troisième non-standard :

- **float** : réel en virgule flottante codé sur 4 octets. Sa plage de valeur est de [+/-]701 411 x 10^{-38} à [+/-]701 411 x 10^{38}
- **double :** réel en virgule flottante codé sur 8 octets. Sa plage de valeur est de [+/-]1.0 x 10^{-307} à [+/-]1.0 x 10^{307}
- **long double** :Réel en virgule flottante codé sur 12 octets. Sa plage de valeur est de [+/-]3.4 x 10^{-4932} à [+/-]1.1 x 10^{4932}. Mais ne faisant pas partie de la norme ANSI, il est souvent transformé en "**double**" par les compilateurs normalisés.

c) Les booléens

Il n'existe pas en langage C de type booléen. Mais la valeur zéro est considérée par le langage Comme "faux" et toute autre valeur différente de zéro est considérée comme "vrai".

2) Les constantes

Les constantes permettent de symboliser les valeurs numériques et alphabétiques de la programmation courante. Il peut paraître trivial d'en parler tellement on y est habitué mais cela est utile en langage C car il existe plusieurs formats de notation.

☞ constantes numériques en base 10 : on les note de la même façon que dans la vie courante. Ex : 5 (cinq) ; -12 (moins douze) ; 17 (dix-sept) ; -45 (moins quarante cinq) ; etc.

☞ constantes numériques en base 8 : on les note en les faisant toutes commencer par le chiffre "***0***". Ex : 05 (cinq) ; -012 (moins dix) ; 017 (quinze) ; -045 (moins trente-sept) ; etc.

☞ constantes numériques en base 16 : on les note en les faisant toutes commencer par les caractères "***0x***" ou "***0X***". Ex : 0x5 (cinq) ; -0x12 (moins dix-huit) ; 0x17 (vingt-trois) ; -0x45 (moins soixante-neuf) ; etc.
Remarque : il est possible de demander explicitement le codage des constantes précédemment citées sur un format "long" en les faisant suivre de la lettre "***l***" ou "***L***". Ex : 5L (nombre "cinq" codé sur 4 octets).

☞ constantes ascii : on les notes en les encadrant du caractère "**'**" (guillemet simple ou accent aigu). Comme il s'agit d'un code ascii, le langage les remplacera par leur valeur prise dans la table des codes ascii : Ex : 'a' (quatre-vingt dix sept) ; 'A' (soixante-cinq) ; '5' (cinquante-trois) ; etc.

☞ constantes code ascii : permettent de coder une valeur ascii ne correspondant à aucun caractère imprimable. On utilise alors un backslash "****" suivi de la valeur ascii convertie en base 8 sur trois chiffres (en complétant avec des zéros si c'est nécessaire) ; ou bien la valeur ascii convertie en base 16 et précédé du caractère "**x**" ; le tout encadré des caractères "**'**" (guillemet simple ou accent aigu). Ex : '\141' (quatre-vingt dix sept ; code ascii de "**a**"), '\x35' (cinquante-trois ; code ascii de "**5**") ; etc.

☞ constantes en virgule flottante
- ☞ notation anglo-saxonne : un nombre avec un "**.**" (point) séparant la partie entière de la partie fractionnelle. Ex : 3.1416
- ☞ notation scientifique : un nombre avec un "***e***" ou un "***E***" indiquant l'exposant du facteur "10". Ex : 3e18 (3×10^{18}).

Remarque : toutes les constantes en virgule flottantes sont codées en format "double". Il est cependant possible de demander explicitement le codage de ces constantes sur un format "float" en les faisant suivre de la lettre "***f***" ou "***F***". Ex : 3.1416F (nombre "3.1416" codé sur 4 octets).

☞ constantes prédéfinies : il s'agit de constantes prédéfinies par le compilateur et ayant une fonction spéciale. On utilise la constante telle qu'elle encadrée des caractères "**'**" (guillemet simple ou accent aigu).

Constante	Signification	Valeur
\n	Fin de ligne	10
\t	Tabulation horizontale	99
\v	Tabulation verticale	11
\b	Retour arrière	8
\r	Retour chariot	13
\f	Saut de page	12
\a	Signal sonore	7
\\	Anti slash	92
\"	Guillemet	34
\'	Apostrophe	44

Remarque : Comme le langage ramène toutes ces constantes vers un codage binaire unique et commun, il est possible de les mélanger à loisir : **'A' + 1 ⇔ 66 ⇔ 0102 ⇔ 0x42 ⇔ 'B'** !!!

3) Les instructions

Le corps d'un programme est composé d'instructions. Ces instructions peuvent être de deux types :

- les instructions simples.
- les instructions composées.

Les instructions simples sont terminées par un **point virgule**. On peut les écrire sur plusieurs lignes en inhibant la fin de ligne par un anti slash (déconseillé). Elles correspondent à des ordres donnés au langage (déclarations de variables, utilisations de variables, appels de fonctions).
Exemple :

```
int a=5;                    // Instruction simple pour déclarer "a"
int b=6;                    // Instruction simple pour déclarer "b"
int c;                      // Instruction simple pour déclarer "c"

c=a + b;                    // Instruction simple pour utiliser "a", "b" et "c"
printf(…);                  // Instruction simple pour appeler la fonction "printf()"
```

Les instructions composées sont des séquences d'instructions simples encadrées par des accolades. Il s'agit en fait de la notion de **bloc**.
Exemple :

```
{                           // Début du bloc d'instructions
    int a=5;                // Instruction simple du bloc
    int b=6;                // Instruction simple du bloc
    int c;                  // Instruction simple du bloc

    c=a + b;                // Instruction simple du bloc
    printf(…);              // Instruction simple du bloc
}                           // Fin du bloc d'instructions
```

4) Les opérateurs

Les opérateurs permettent de manipuler les variables et les constantes. Ils sont de trois type :

- opérateurs unaires : ils prennent en compte un seul élément
- opérateurs binaires : ils prennent en compte deux éléments
- opérateurs ternaires : ils prennent en compte trois éléments

a) Opérateurs binaires

Ce sont les plus simples à appréhender. Les premiers peuvent s'appliquer à tout type de variable ou constante (entière ou à virgule flottante) :

- Egal (=) : Affectation d'une valeur à une variable. Ex : a = 5
- Plus (+) : Addition de deux valeurs. Ex : 13 + 3
- Moins (-) : Soustraction de deux valeurs. Ex : 13 – 3
- Multiplié (*) : Multiplication de deux valeurs. Ex : 13 * 3
- Divisé (/) : Division de deux valeurs. Ex : 13 / 3
- Test d'égalité (==) : Permet de vérifier l'égalité entre deux valeurs. Ex : a == b (si "a" est égal à "b" alors l'expression vaudra une valeur quelconque différente de "0" symbolisant le booléen "vrai" ; sinon l'expression vaudra "0"). **Attention** : Ecrire "=" en pensant "==" peut produire des incohérences graves dans le programme qui se compilera cependant sans erreur !
- Test de différence (!=) : Permet de vérifier l'inégalité entre deux valeurs. Ex : a != b (si "a" est différent de "b" alors l'expression vaudra une valeur quelconque différente de "0" symbolisant le booléen "vrai" ; sinon l'expression vaudra "0"). Cet opérateur est l'opposé du "==".
- Tests d'inégalités diverses (>, <, >=, <=) : Permet de vérifier les grandeurs relatives entre deux valeurs. Ex : a >= b (si "a" est supérieur ou égal à "b" alors l'expression vaudra une valeur quelconque différente de "0" symbolisant le booléen "vrai" ; sinon l'expression vaudra "0").

Les autres ne peuvent s'appliquer qu'aux valeurs (variables ou constantes) de type entière (char, short, long, int) :

- Modulo (%) : Reste d'une division entière entre deux valeurs. Ex : 13 % 3
- "ET" logique (&&) : Opération booléenne "ET" entre deux booléens. Ex : a && b (si "a" est "vrai" **ET** "b" est "vrai", alors l'expression vaudra une valeur quelconque différente de "0" symbolisant le booléen "vrai" ; sinon l'expression vaudra "0").
- "OU" logique (||) : Opération booléenne "OU" entre deux booléens. Ex : a || b (si "a" est "vrai" de "0" **OU** "b" est "vrai", alors l'expression vaudra une valeur quelconque différente de "0" symbolisant le booléen "vrai" ; sinon l'expression vaudra "0").

Les suivants ne peuvent s'appliquer qu'aux valeurs (variables ou constantes) de type entière (char, short, long, int) et agiront individuellement sur chaque bit de la constante ou variable.

- "ET" bit à bit (&) : Opération booléenne "ET" appliquée pour chaque bit des deux valeurs ; le $n^{ième}$ bit de la première valeur étant comparé au $n^{ième}$ bit de la seconde valeur. Ex : a & 5. Cette expression vaudra "5" si le premier et troisième bits de "a" sont à "1" (masque).
- "OU" bit à bit (|) : Opération booléenne "OU" appliquée pour chaque bit des deux valeurs ; le $n^{ième}$ bit de la première valeur étant comparé au $n^{ième}$ bit de la seconde valeur. Ex : a | 5. Cette expression donnera une valeur où les premier et troisième bits seront toujours à "1".
- "OU EXCLUSIF" bit à bit (^) : Opération booléenne "OU EXCLUSIF" appliquée pour chaque bit des deux valeurs ; le $n^{ième}$ bit de la première valeur étant comparé au $n^{ième}$ bit de la seconde valeur. Ex : a ^ 5. Cet expression donnera une valeur où les premier et troisième bits seront à l'inverse de ceux de "a" (bascule).
- Décalage à droite (>>) : Chaque bit de la valeur sera décalé vers la droite de "x" positions. Ex : 12 >> 3. Cela donne le même résultat que de diviser par 2^3.
- Décalage à gauche (<<) : Chaque bit d'une valeur sera décalé vers la gauche de "x" positions. Ex : 12 << 4. Cela donne le même résultat que de multiplier par 2^4.

b) Opérateurs d'opération avec affectation

Chaque opérateur binaire précédemment décrit peut être associé à l'opérateur "=" d'affectation. Cela permet de raccourcir une expression du style "a = a + x" par l'expression "a+=x" (sans espace entre le "+" et le "=").

c) Opérateurs unaires

Une fois que l'on a compris les opérateurs binaires, les opérateurs unaires viennent plus facilement.
Les premiers peuvent s'appliquer à tout type de variable ou constante (entière ou à virgule flottante) :

- Moins unaire (-) : Fait passer en négatif un nombre positif et inversement. Ex : -5
- Plus unaire (+) : Juste pour assurer la cohérence avec la vie réelle.Ex : +5
- Négation logique (!) : Inverse les booléens "vrai" et "faux". Ex : !5
- Taille de (sizeof) : Renvoie la taille en octet de la valeur ou du type demandé. Ex : sizeof(char)

Le suivant ne peut s'appliquer qu'aux valeurs (variables ou constantes) de type entière (char, short, long, int) et agira individuellement sur chaque bit de la constante ou variable.

- "NOT bit à bit" (~) : Opération booléenne "NOT" appliquée à chaque bit de la valeur.. Ex : ~a. Cet expression donnera une valeur où tous les bits seront à l'inverse de ceux de "a".

Les quatre derniers ne s'appliquent qu'aux variables de type entière (char, short, long, int). Pour chaque exemple, nous prendrons une variable "i" initialement affectée avec "5".

- Pré-incrémentation (++ placé avant la variable) : Ex : ++i ("i" commence par s'incrémenter et passe à "6". Ensuite, l'expression renvoie la valeur finale de "i" ; soit "6").
- Post-incrémentation (++ placé après la variable) : Ex : i++ (l'expression renvoie d'abord la valeur initiale de "i" ; soit "5". Ensuite, "i" s'incrémente et passe à "6").
- Pré-décrémentation (-- placé avant la variable) : Ex : --i ("i" se commence par se décrémenter et passe à "4". Ensuite, l'expression renvoie la valeur finale de "i" ; soit "4").
- Post-décrémentation (-- placé après la variable) : Ex : i-- (l'expression renvoie d'abord la valeur initiale de "i" ; soit "5". Ensuite, "i" se décrémente et passe à "4").

d) Opérateurs ternaire

Il n'y en a qu'un seul qui peut s'appliquer à tout type de valeur

- vrai…alors…sinon (? :) : Ex : x ?y :z (si "x" est différent de "0" donc "vrai" ; alors l'expression complète vaut "y" sinon elle vaut "z").

e) Valeur et renvoi d'instruction

Toute instruction renvoie une valeur. Exemple, l'instruction "5;" renvoie une valeur qui est… "5". Il est possible, pour ne pas la perdre, de récupérer cette valeur par l'opérateur d'affectation "=" dans l'instruction "a=5;". Mais cette dernière instruction renvoie aussi une valeur qui est… "5". Il est alors encore possible de la récupérer par un nouvel opérateur d'affectation "=" dans l'instruction "b=a=5" ; etc. Il est ainsi possible de récupérer la valeur de chaque instruction écrite. Ex : a=(b == 2) (récupère dans "a" le booléen "vrai" ou "faux" résultant de la comparaison entre "b" et "2").

f) Opérateur de concaténation

Celui-ci ne s'applique qu'aux instructions

- Concaténation d'instruction (,) : Permet de mettre plusieurs instructions à suivre et de ne prendre que la valeur de la dernière. Ex : int a,b,c=5 (déclare trois variables "a", "b" et "c" de type "int" et affecte "5" à "c").

g) Exercices

Calculez les résultats des expressions suivantes :

```
(2 + 3) * 4                          // Réponse : 20 (2+3 = 5 puis 5 * 4)
(2 << 4) + 3                         // Réponse : 35 (2 * 16 = 32 puis 32 + 3)
1 | 3                                // Réponse : 3 (0x01 | 0x11)
(2 << 1) > 3 ?25 >> 2 : 25 >> 1      // Réponse : 6 (2 * 2 = 4 donc 25 / 4 = 6)
4 > 5                                // Réponse : 0 ("faux")
2,3,4+5                              // Réponse : 9 (2 et 3 ne sont pas traités)
```

L'ensemble complet des opérateurs avec leurs priorités est mis en annexe.

5) L'affichage – La saisie

L'affichage d'une valeur (variable ou constante) se fait en utilisant la fonction "***printf()***" qui signifie "print formating"
Cette fonction s'utilise de la manière suivante :

☞ Premier argument et le plus complexe : Message à afficher. Ce message, encadré par des **"** (double guillemets) constitue en fait tout le texte que le programmeur désire afficher. A chaque position du texte où il désire que soit reporté une variable ou une valeur d'expression, il doit l'indiquer en positionnant un signe "***%***" (pour cent) suivi d'un caractère indiquant la manière dont il désire afficher ladite valeur.

☞ Autres arguments : Expressions et variables dont on désire afficher la valeur.

Exemple :

```
main()
{
        int a=5;
        int b=6;
        printf("Le résultat de %d ajouté à %d plus 1 est %d", a, b, a + b + 1);
}
```

La saisie d'une variable se fait en utilisant la fonction "***scanf()***" qui signifie "scan formating"
Cette fonction s'utilise de la manière suivante :

☞ Premier argument et le plus complexe : Masque de saisie. Ce masque, encadré par des **"** (double guillemets) constitue en fait tout ce que devra taper l'utilisateur quand il saisira ses variables. A chaque position du texte où le programmeur désire que soit reporté une variable à saisir, il doit l'indiquer en positionnant un caractère "***%***" (pour cent) suivi d'un caractère indiquant la manière dont il désire que soit saisie ladite valeur. Bien souvent, il n'y a aucun masque de saisie car c'est très contraignant.

☞ Autres arguments : Variables à saisir, **chacune précédée du caractère "&"**

Exemple :

```
main()
{
        int jj;
        int mm;
        int aa;
        printf("Saisissez une date sous la forme jj/mm/aa  :\n");
        scanf("%d/%d/%d", &jj, &mm, &aa);
}
```

Erreur fréquente :

```
main()
{
        int i;

        scanf("Saisissez un nombre : %d", &i);          // Erreur => scanf n'affiche rien

        // Une invite s'affiche avec "printf"
        /* Il faut écrire :
                printf(" Saisissez un nombre :\n");
                scanf("%d", &i);
        */
}
```

L'ensemble des options de saisie et d'affichage sont mises en annexe.

6) La conversion de type (casting)

a) Généralités

Lorsque le langage doit faire un calcul avec des valeurs numériques ; ce calcul se fait dans la plus grande précision des nombres intervenants ; et non dans la précision du résultat attendu.
Exemple : float f=20 / 7;
Résultat :"f" vaut "2.0000" alors qu'il était sûrement attendu "2,8571". Mais "20" et "7" étant une écriture typique de nombres au format "int", le calcul a été entièrement fait en codage "int". Pour obtenir le bon résultat, il aurait fallu écrire au-moins une des deux constante dans une notation en virgule flottante en écrivant par exemple "float f = 20.0 / 7" ou bien "float f = 20 / 7e0".

b) Conversions de type explicites

Le problème précédent reste le même si on opère avec des variables de type "int" pour mettre le résultat dans un "float". Le résultat récupéré dans le "float" ne correspondra pas à la valeur attendue.
Exemple : int a=20, b=7; float f=a / b;

La seule façon d'avoir un résultat dans "f" correct est de forcer au moins une des deux variables à être considérée comme un "float". Cette opération se nomme "casting" (du verbe anglais "to cast" qui signifie "jouer un rôle").
On crée un "casting" sur une variable en indiquant, avant la variable et entre parenthèses, dans quel type elle doit être considérée.
Exemple :

```
int a=20;
int b=7;
float f;

f=(float)a / b;          // Le calcul est effectué en considérant "a" comme un "float"
f=a / (float)b;          // Le calcul est effectué en considérant "b" comme un "float"
f=(float)a / (float)b;   // Les deux variables "a" et "b" sont considérées comme des "float"
// Dans ces trois écritures, on aura bien "f=2,8571"
```

c) Conversions de type implicites

Les conversions de type implicites se font de façon naturelle chaque fois que des variables de types différents sont utilisées dans une opération. A ce moment là, la variable de type le plus "étroit" est convertie dans la variable de type le plus "large" et le "cast" n'est pas nécessaire.

```
int a=20;
float b=7.0;
double f=a / b;                  // Conversion implicite de "a" en "float"
```

On peut aussi copier une variables d'un type dans une variable d'un autre type. Mais si la variable réceptrice est plus "étroite", il y a perte de données.

Conversion automatique sans perte de précision et sans perte de signe

char	⇒	**int**	le caractère est placé dans l'octet le moins significatif
short	⇒	**long**	expansion du signe
int	⇒	**float**	pas de problème
float	⇒	**double**	pas de problème

Conversions automatiques avec perte de précision

int	⇒	**char**	perte des octets les plus significatifs
long	⇒	**short**	perte des octets les plus significatifs
float	⇒	**int**	perte de la partie décimale
double	⇒	**float**	perte de précision

IV) LES TYPES DERIVES

1) Les tableaux

Chaque type vu précédemment peut être déclaré sous forme de tableau en faisant suivre le nom de la variable de crochets "***[…]***" et en indiquant entre les crochets le nombre d'éléments du tableau. On aura alors en mémoire plusieurs zones contiguës réservées pour le tableau et chaque zone aura la taille d'un élément du tableau. La contiguïté des zones sera importante lorsqu'on parlera des pointeurs.
L'initialisation peut se faire lors de la déclaration en indiquant les valeurs mises entre accolades "***{…}***" et en séparant chaque valeur par le caractère "**,**" (virgule)
Les tableaux peuvent être sur plusieurs dimensions en mettant plusieurs séries de crochets d'affilées. Si on initialise un tableau lors de sa déclaration, on peut omettre la taille de sa première dimension, le compilateur remplira de lui-même le vide en fonction des valeurs qu'il doit initialiser.
Exemple :

```
main()
{
        int i[10];                                      // Tableau de 10 entiers

        double d[20];                                   // Tableau de 20 doubles

        char voyelle[]={'a', 'e', 'i', 'o', 'u', 'y'};  // Tableau de 6 caractères ("[6]" implicite)

        char semaine[7][8]={                            // Tableau de 7 lignes sur 8 colonnes
                {'l', 'u', 'n', 'd', 'i'},              // La taille "7" de la première dimension
                {'m', 'a', 'r', 'd', 'i'},              // est facultative. On aurait pu mettre
                {'m', 'e', 'r', 'c', 'r', 'e', 'd', 'i'},   // aussi "char semaine[][8]={…}"
                {'j', 'e', 'u', 'd', 'i'},
                {'v', 'e', n', 'd', 'r', 'e', 'd', 'i'},
                {'s', 'a', m', 'e', 'd', 'i'},
                {'d', 'i', 'm', 'a', 'n', 'c', 'h', 'e'}
        };                                              // Fin du tableau (plus lisible ainsi)
…
```

L'accès à un élément du tableau se fait en donnant le nom du tableau suivi de l'indice voulu entre crochets. Cet indice peut lui-même être une variable ou une expression. Attention, en C, un tableau commence à l'indice "**0**". De plus, il n'est fait **aucun contrôle** sur la cohérence de l'indice (contrôle de dépassement par exemple). C'est au programmeur de vérifier la bonne validité de l'indice par rapport à la taille du tableau !
Exemple (suite) :

```
…
        int j=3;
        i[-1]=i[42]=0;              // Autorisé par le compilateur malgré l'indice incohérent

        printf("la première voyelle est %c, la quatrième voyelle est %c\n", voyelle[0], voyelle[j]);

        printf("la première lettre du dernier jour de la semaine est %c\n", semaine[6][0]);
}
```

Un tableau n'étant pas un type de base du langage, il ne peut pas être manipulé (copié ou comparé) comme une variable simple. En particulier, la syntaxe "tab={…}" n'est autorisée que lors de la déclaration du tableau. Si on désire le remplir en dehors de sa déclaration il faut alors travailler dans le tableau élément par élément avec une boucle.

2) Exercice

Ecrire un programme qui fasse saisir un tableau de 5 nombres à virgule flottante.
Ensuite, il les affiche

```
main()
{
	// Déclaration des variables
	float tab[5];					// Tableau des nombres à saisir

	// Saisie
	printf("Saisissez le premier nombre  :");
	scanf("%f", &tab[0]);

	printf("Saisissez le second nombre  :");
	scanf("%f", &tab[1]);

	printf("Saisissez le troisième nombre  :");
	scanf("%f", &tab[2]);

	printf("Saisissez le quatrième nombre  :");
	scanf("%f", &tab[3]);

	printf("Saisissez le cinquième nombre  :");
	scanf("%f", &tab[4]);

	// Affichage
	printf("le premier nombre saisi est %f noté aussi %e\n", tab[0], tab[0]);
	printf("le second nombre saisi est %f noté aussi %e\n", tab[1], tab[1]);
	printf("le troisième nombre saisi est %f noté aussi %e\n", tab[2], tab[2]);
	printf("le quatrième nombre saisi est %f noté aussi %e\n", tab[3], tab[3]);
	printf("le cinquième nombre saisi est %f noté aussi %e\n", tab[4], tab[4]);

	// Evidemment, cela aurait été plus simple avec une boucle...
}
```

3) Les chaînes de caractères

Une chaîne de caractère n'existe pas en C. C'est à dire que le langage ne connaît pas le type "chaîne". Les programmeurs ont alors adopté la convention suivante : **une chaîne de caractères sera représentée par un tableau de caractères (chaque lettre de la chaîne occupant un caractère du tableau) terminé par un caractère de valeur "0"** (rappelons qu'un caractère n'est qu'un nombre entier codé sur 8 bits et donc pouvant prendre une valeur comprise entre –128 et 127; ou entre 0 et 255). Bien souvent, cette valeur sentinelle terminant une chaîne est notée dans le format "constante code ascii 0" ('\0').

Il en résulte les corollaires suivants :

- tout tableau de caractères contenant un caractère '\0' pourra être traité comme une chaîne de caractères par les fonctions de manipulation de chaînes.
- toute fonction de création ou de remplissage de chaînes rajoutera toujours un caractère '\0' en fin de tableau.

Evidemment, il appartient au programmeur, qui va créer un tableau destiné à stocker une chaîne, de prévoir un emplacement supplémentaire pour stocker la valeur "0".
Exemple :

```
char jour[6]={'l', 'u', 'n', 'd', 'i', '\0'};
```

Cette notation étant lourde, le compilateur autorise d'écrire la chaîne d'un seul bloc entre doubles quottes sans écrire le "0" final. Mais il ne faut pas oublier que ce n'est qu'une autre façon d'écrire et que, même si on ne l'écrit plus, le "0" final reste présent donc on doit en tenir compte dans la place réservée au tableau.
Exemple :

```
char jour[6]="lundi";                    // Ce tableau contient 5 lettres plus la valeur "0"
```

Attention aux erreurs de programmation suivantes :

- Si un caractère '\0' vient se mettre au milieu d'une chaîne de caractères, cette dernière sera considérée ensuite comme tronquée car toutes les fonctions de manipulation de chaînes s'arrêteront au premier '\0' rencontré.
- Si le caractère '\0' disparaît du tableau de caractères; les fonctions de manipulation de chaînes ne pourront plus repérer la fin de chaîne et risquent de boucler indéfiniment dans leur traitement.

La fonction "***printf()***" offre au programmeur la possibilité d'afficher une chaîne d'un seul coup sans avoir à programmer de boucle car celle-ci a été programmée dans la fonction. Il suffit de mettre l'option "%s" et de ne donner que le nom du tableau sans crochet.
La fonction "***scanf()***" offre au programmeur la possibilité de saisir une chaîne d'un seul coup sans avoir à programmer de boucle car celle-ci a été programmée dans la fonction. Il suffit de mettre l'option "%s" et de ne donner que le nom du tableau sans crochet ni caractère "&".

Exemple :

```
scanf("%s", jour);
printf("Le jour saisi est %s\n", jour);
```

Comparer deux tableaux ne peut se faire qu'en comparant chaque élément des deux tableaux avec une boucle.
N'écrivez donc pas *if (jour == "lundi")* car cette instruction est possible mais elle ne fait que comparer les adresses mémoires des deux tableaux (*cf. chapitre sur les pointeurs*).

4) La création de nouveau type

Il est possible, en C, de créer un nouveau type à partir d'un type existant en utilisant l'instruction "typedef".

```
typedef type_existant nouveau_type;
```

Ceci offre en fait une souplesse en ce qui concerne la maintenance. Imaginons qu'un programme travaille sur un tableau de moins de 250 éléments. Tous les indices du tableau peuvent être définis comme des "*unsigned char*" puisque ce format permet de coder les nombres entiers jusqu'à 255. Mais si le tableau change de taille et dépasse 256 éléments, il faudra reprendre tout le programme et modifier les types de tous les indices.
En utilisant l'instruction "*typedef*", il est possible de créer un alias sur le type "*unsigned char*" et de définir tous les indices sur cet alias. Si le tableau change de taille, il suffit de changer l'alias.

Exemple1 (tableau < 256 éléments) :

```
typedef unsigned char t_indice;
main()
{
        // Déclaration des variables
        float tableau[250];
        t_indice i;

        // Balayage du tableau
        for (i=0; i < 250; i++)
                tab[i]=0;
}
```

Exemple2 (tableau > 256 éléments) :

```
typedef unsigned short t_indice;
main()
{
        // Déclaration des variables
        float tableau[5000];
        t_indice i;

        // Balayage du tableau
        for (i=0; i < 5000; i++)
                tab[i]=0;
}
```

V) LES INSTRUCTIONS DE CONTROLES

Une instruction de contrôle est une instruction qui permet de rompre l'exécution séquentielle du programme en fonction de différents critères.

1) L'instruction "if ()… else"

Cette instruction permet une exécution conditionnelle d'une partie du code.
Syntaxe :

```
if (X)                                  // Attention, pas de "point-virgule" !!!
        Y;
else                                    // Facultatif
        Z;
T;
```

Si l'instruction "X" est "vrai" ; c'est à dire si elle renvoie une valeur différente de "0" ; alors le langage ira exécuter l'instruction "Y". Sinon ; et à condition qu'il y ait une instruction "else" ; le langage ira exécuter l'instruction "Z".
Dans tous les cas, le langage ira exécuter l'instruction "T".

La première remarque que l'on peut faire est qu'une structure "if" ne permet qu'une seule instruction pour chaque cas "vrai" ou "faux". Mais rien n'empêche que cette instruction soit complexe… donc un bloc !

```
if (X)
{                                       // Remarquez le début de bloc
        Y1;
        Y2;
        Y3.
}                                       // Ici, on a la fin du bloc/
else
{                                       // Remarquez aussi le début de bloc
        Z1;
        Z2;
        Z3.
}                                       // Ici, on a encore la fin du bloc
T;
```

La seconde remarque est le contenu des parenthèses ! Il faut que cette instruction renvoie une valeur. Or, toute instruction renvoie une valeur. Par exemple, l'instruction "5" renvoie la valeur "5" qui peut être récupérée dans une variable "a" dans la syntaxe "a=5". Ainsi, toute instruction peut être testée par un "if".

```
if (5)                                  // Idiot car toujours vrai
        printf(…);

if (0)                                  // Idiot car toujours faux
        printf(…);                      // Jamais exécuté/

if (var=5)                              // Affecte "5" à "var" et renvoie "5" donc "vrai"
        printf(…);
/*      Peut-être que le programmeur pensait "si var est égal à 5" ?
        Auquel cas il aurait fallu écrire "if (var == 5)" (avec deux signes "=" collés) */

if (var)                                // Si "var" est "vrai" donc "différent de "0"
        printf(…);
```

2) L'instruction "while ()"

Cette instruction permet une exécution un certain nombre de fois (boucle "tant que").
Syntaxe :

```
while (X)                                   // Attention, pas de "point-virgule" !!!/
        Y;
Z;
```

Tant que l'instruction "X" est "vrai" ; c'est à dire tant qu'elle renvoie une valeur différente de "0" ; alors le langage exécutera l'instruction "Y".
Dès que la l'instruction "X" devient fausse, le langage sort de la boucle et passe à l'instruction "Z".

Même remarque pour le "if" ; la structure "while" ne permet de boucler que sur une seule instruction. Mais ici aussi rien n'empêche que cette instruction soit un bloc entre accolades.

```
while (X)
{                                           // Remarquez le début de bloc
        Y1;
        Y2;
        etc.
}                                           // Ici, on a la fin du bloc
Z;
```

Il en va de même pour l'instruction entre parenthèses. C'est la valeur résiduelle de l'instruction qui est évaluée.

```
while (5)                                   // Toujours vrai donc boucle infinie
        printf(…);

while (0)                                   // Toujours faux donc pas de boucle (utilité ?)
        printf(…);                          // Jamais exécutée

while (var=5)                               // Affecte "5" à "var" et renvoie "5" donc "vrai"
        printf(…);
/*      Probablement écrit cela en pensant "tant que "var" est égal à "5" ?
        Alors il faut écrire "while (var == 5)"
        Cette erreur est très fréquente chez les débutants)             */

while (var)                                 // Tant que "var" est "vrai" donc "différent de "0"
        printf(…);
```

3) L'instruction "do… while ()"

Cette instruction permet une exécution un certain nombre de fois (boucle "tant que") mais l'évaluation de la condition de boucle se faisant en fin de boucle, celle-ci est au moins exécutée une fois.
Syntaxe :

```
do                                  // Attention, pas de "point-virgule"
        X;                          // Exécuté au-moins une fois
while (Y);                          // Attention, il faut un "point-virgule" !!!
Z;
```

Le langage exécute l'instruction "X" puis.remonte en début de boucle tant que l'instruction "Y" est "vrai".
Dès que la l'instruction "Y" devient fausse, le langage sort de la boucle et passe à l'instruction "Z".

Même remarque pour le "if" et le "while" précédent ; la structure "while" ne permet de boucler que sur une seule instruction. Mais ici aussi rien n'empêche que cette instruction soit un bloc entre accolades.

```
do
{                                   // Remarquez le début de bloc
        X1;
        X2;
        etc.
} while (Y);                        // Ici, on a la fin du bloc
Z;
```

Il en va de même pour l'instruction entre parenthèses. C'est la valeur résiduelle de l'instruction qui est évaluée.

```
do
        printf(…);
while (5);                          // Toujours vrai donc boucle infinie


do
        printf(…);                  /* Exécuté une fois
while (0);                          /* Toujours faux donc pas de boucle (utilité ?)


do
        printf(…);
while (var=5);                      // Toujours la même erreur : Ecrire "=" en pensant à "=="

do
        printf(…);
while (var);                        // Tant que "var" est "vrai" donc "différent de "0"
```

4) L'instruction "for ()"

Cette instruction permet de rassembler dans une seule ligne une initialisation, une condition de continuation de boucle et un incrément.

Syntaxe :

```
for (X; Y; Z)
// Attention, des "points-virgule" entre chaque instruction mais pas en fin de for !!!
        T;
Q;
```

Le langage ira exécuter l'instruction "X" avant la boucle. Celle-ci est souvent une initialisation de variable. Ensuite, tant que l'instruction "Y" est "vrai" ; c'est à dire tant qu'elle renvoie une valeur différente de "0" ; alors le langage exécutera l'instruction "T". En général, l'instruction "Y" contrôle la valeur de la variable. Enfin, il exécute l'instruction "Z" qui est souvent un incrément de la variable.
Dès que la l'instruction "Y" devient fausse, le langage sort de la boucle et saute l'instruction "Z" pour passer directement à l'instruction "Q".

Même remarque pour les autres structures ; la structure "for" ne permet de boucler que sur une seule instruction qui peut être un bloc entre accolades. Par ailleurs, on peut toujours remplacer un "for" par un "while"

```
// Exemple avec "for"
for (X; Y; Z)
{                                       // Remarquez le début de bloc
        T1;
        T2;
        etc.
}                                       // Ici, on a la fin du bloc
Q;

// Remplacement par un "while"
X;
while (Y)
{                                       // Remarquez quand-même le début de bloc
        T1;
        T2;
        etc.
        Z;
}                                       // Ici, on a la fin du bloc
Q;
```

Exemples :

```
// Incrément simple
for (i=0; i < 10; i++)
        printf("%d\n", i);

// Incrément double (remarquez l'utilisation judicieuse de la virgule dans le "for")
for (i=0, j=10; i < 10; i++, j++)
        printf("%d %d\n", i, j);

// Erreur fréquente dans la programmation d'une boucle "for"
for (i=0; i == 10; i++)
/* L'erreur est de penser jusqu'à i égal à 10 alors que le langage lit tant que i égal à 10 !!! */
        printf("%d\n", i);
// La syntaxe "for (i=0; i = 10; i++)" n'est pas non plus correcte !!!
```

5) Les instructions "break" et "continue"

L'instruction "***break***" permet au langage d'interrompre une boucle sans retourner évaluer sa condition. Dans le cas d'une boucle "for", l'instruction "break" fait aussi éviter la troisième instruction de la parenthèse (instruction "Z").
L'instruction "***continue***" permet au langage de remonter à l'évaluation de la condition de boucle. Bien entendu, si cette condition n'est plus remplie, le langage sort de la boucle.

Ces deux instructions sont contraires à la philosophie de "programmation structurée", mais elles évitent beaucoup de lourdeurs d'écritures.

Exemple sans "break" :

```
// Boucle de 0 à 9
i=0;
while (i < 10)
{
        if (problème quelconque)
        {
                printf("Problème\n");
                i=12 ;
        }
        else
                printf("%d\n", i);
        i++;
}
```

Exemple avec "break" :

```
// Boucle de 0 à 9
i=0;
while (i < 10)
{
        if (problème quelconque)
        {
                printf("Problème\n");
                break;
        }
        printf("%d\n", i);
        i++;
}
```

Exemple sans "continue" :

```
// Boucle de 0 à 9
i=0;
while (i < 10)
{
        if (problème quelconque)
                printf("Problème\n");
        else
        {
                printf("%d\n", i);
                i++;
        }
}
```

Exemple avec "continue" :

```
// Boucle de 0 à 9
i=0;
while (i < 10)
{
        if (problème quelconque)
        {
                printf("Problème\n");
                continue;
        }
        printf("%d\n", i);
        i++ ;
}
```

6) L'instruction "goto"

Cette instruction permet de se positionner n'importe où dans le source et de continuer l'exécution à partir du nouveau positionnement. Elle est toutefois assez décriée par une grande partie des programmeurs qui respectent le langage et sa philosophie. Cependant une utilisation fine et réfléchie permet parfois la simplification d'un code sinon trop lourd (comme une gestion d'erreurs par exemple).

Syntaxe :

```
etiquette:                          // Remarquez le caractère "double-point" !!!
for (…)
        X;
if (…)
        Y ;
etc.
goto etiquette;                     // Pour se placer sur l'étiquette
```

7) L'instruction "switch ()… case"

Cette instruction permet un branchement sur cas multiples.
Syntaxe :

```
switch (X)                          // Attention, pas de "point-virgule"
{                                   // Attention, une "accolade" (début de bloc) obligatoire !!!
        case <constante 1> :
                Y;
                Z;
                break;
        case <constante 2> :
                P;
                Q;
                break;
        etc.
        default :
                R;
                S;
}                                   // Fin du bloc
```

Le langage évalue l'instruction "X". Selon sa valeur, il se positionne sur la constante qui lui correspond et exécute les instructions qui s'y trouvent. Sinon, il se positionne dans le cas "default" qui est facultatif et qui correspond aux cas non prévus.
Le terme "<constante>" signifie qu'il est impossible d'y mettre une variable. Le programmeur désirant utiliser un algorithme de ce type devra alors passer par des instructions "if ()… else".

Remarque : Le compilateur transforme littéralement une instruction "*switch()*" en suite d'instructions "*goto*" (remarquez le caractère "double point" terminant chaque constante lui donnant ainsi le rôle d'une étiquette). Cela implique qu'une fois positionné sur l'étiquette correspondante, le programme déroule et exécute ensuite linéairement toutes les instructions suivantes y compris les instructions correspondantes à d'autres cas.
Cette attitude peut être heureusement inhibée par l'emploi judicieux de l'instruction "***break***" à la fin de chaque cas (inutile malgré tout pour le cas "*default*").

Exemple :

```
main()
{
        int nb;

        printf("Saisissez un nombre  :");
        scanf("%d", &nb) ;

        switch (nb)
        {
                case 0 : // Sera exécuté si "nb" vaut 0
                        printf("Valeur zéro\n");
                case 2 : case 4 : case 6 : case 8 :
                        // Sera exécuté si "nb" vaut 2, 4, 6, 8 et aussi 0 car il n'y a pas de break au cas 0
                        printf("Nombre %d pair infèrieur à 10\n", nb);
                        break;
                case 1 : case 3 : case 5 : case 7 : case 9 :
                        // Sera exécuté si "nb" vaut 1, 3, 5, 7, 9
                        printf("Nombre %d impair infèrieur à 10\n", nb);
                        break;
                default :
                        printf("Nombre %d trop grand pour être évalué\n", nb);
        }
}
```

8) L'instruction "exit ()"

Cette instruction permet d'arrêter et quitter volontairement le programme en dehors de sa fin naturelle (fin de la fonction "*main*").
Syntaxe :

```
exit(n);
```

La valeur "n" correspond à une valeur numérique qui sera récupérée par le processus ayant lançé le programme. Si ce programme a été lancé depuis un "shell" d'Unix, cette valeur est renvoyée dans la variable shell "***$?***".
Par convention, une valeur à "**0**" indique que le programme s'est déroulé normalement et une **valeur différente de "0"** indique que la sortie est due à une exception (erreur).
Bien que cette instruction puisse être appelée de n'importe quelle fonction, la philosophie du langage veut que seule la fonction "***main()***" utilise l'instruction "***exit()***".

9) Exercices

Ecrire un programme qui fasse saisir un tableau de 10 nombres à virgule flottante.
Ensuite, il les affiche.
Enfin, il recherche et affiche les éléments du tableau de valeur comprise entre "-1" et "+1"

```
main()
{
	// Déclaration des variables
	unsigned short i;			// Indice de boucle dans le tableau
	float tab[10];				// Tableau des nombres à saisir

	// Saisie
	for (i=0; i < 10; i++)			// Le premier indice est à "0"
	{
		printf("Saisissez le nombre %hu :", i + 1);
		// "i + 1" pour faire correspondre l'indice affiché avec la "vision humaine"
		scanf("%f", &tab[i]);
	}

	// Affichage des valeurs du tableau
	for (i=0; i < 10; i++)
		printf("L'élément %hu est %f\n", i + 1, tab[i]);

	// Recherche et affichage des valeurs correspondantes aux critères
	for (i=0; i < 10; i++)
	{
		if (tab[i] >= -1.0 && tab[i] <= 1.0)
			printf("L'élément %hu de valeur %f correspond aux critères\n", i + 1, tab[i]);
	}

	// Même recherche mais en utilisant la condition inverse (juste pour le "fun")
	for (i=0; i < 10; i++)
	{
		if (tab[i] < -1.0 || tab[i] > 1.0)
		// Remarquez que le "ET" est devenu un "OU" (loi de "De Morgan")
			continue;		// On va directement sur "i++"

		// Ici, on est certain que l'élément du tableau correspond aux critères
		printf("L'élément %hu de valeur %f correspond aux critères\n", i + 1, tab[i]);
	}
}
```

Ecrire un programme qui affiche :
XXXXXXXXXX
YXXXXXXXXX
YYXXXXXXXX
YYYXXXXXXX
YYYYXXXXXX
YYYYYXXXXX
YYYYYYXXXX
YYYYYYYXXX
YYYYYYYYXX
YYYYYYYYYX
YYYYYYYYYY
Remarque : Si on numérote chaque ligne en commençant par "0", alors à la ligne "n", il y a "n" **Y** et "10 – n" **X**.

```
main()
{
	// Déclaration des variables
	unsigned short lig;			// Indice de ligne
	unsigned short i;			// Indice de boucle

	// Boucle sur 11 lignes
	for (lig=0; lig < 11; lig++)		// La première ligne est à "0"
	{
		// Boucle sur le nombre de "Y"
		for (i=0; i < lig; i++)
			// Affichage des "Y"
			printf("Y");

		//Boucle sur le nombre de "X"
		for (i=10 - lig; i > 0; i--)
			//Affichage des "X"
			printf("X");
		/*
			On peut mettre plus simplement "for (i=0 ; i < (10 – lig); i++)" mais
			l'opération (10 – lig) est alors exécutée à chaque boucle ce qui les ralentit
                                                                                */

		// Fin de ligne
		printf("\n");
	}
}
```

Ecrire un programme qui affiche :
1
12
123
1231
12312
123123
1231231
etc…
Remarque : Si on numérote chaque ligne en commençant par "0", alors la ligne "n" contient "n + 1" valeurs et s'arrête à "n % 3 + 1".

```
main()
{
	// Déclaration des variables
	unsigned short lig;			// Indice de ligne
	unsigned short i;			// Indice de boucle

	// Initialisation de la ligne
	lig=0;

	// Boucle infinie
	while (1)
	{
		// Boucle sur le nombre de valeurs à afficher
		for (i=0; i <= lig; i++)
			// Affichage des valeurs
			printf("%d", i % 3 + 1);

		// Fin de ligne – Incrément numéro de ligne
		printf("\n");
		lig++;
	}
}
```

VI) LES FONCTIONS

Une fonction est un sous-programme qui reçoit éventuellement des arguments en entrée (ces arguments sont **toujours des valeurs numériques**) et qui retourne éventuellement **une valeur** d'un certain type vers la fonction appelante.

L'écriture d'une fonction se fait en plusieurs étapes :

- On définit ce dont elle a besoin en entrée et ce qu'elle retourne en sortie.
- On écrit le **prototype**. Le prototype indique notamment le nombre et les types des arguments nécessaires ainsi que le type du résultat renvoyé. Le prototype est déclaré **avant** la définition de la fonction. Il permet au compilateur de contrôler la cohérence des appels. Cette étape n'est pas obligatoire mais le compilateur doit connaître la fonction avant qu'elle ne soit appelée. Si le programmeur ne désire pas établir de prototype, il doit coder le corps de la fonction appelée avant de coder le corps de la fonction appelante.
- On écrit enfin le corps de la fonction (on code le traitement attendu).

Cette définition peut être placée dans le même fichier source que la fonction appelante ou dans un autre fichier (mais il faudra indiquer au compilateur quels fichiers utiliser).
Il n'y a pas de possibilité d'imbrication des fonctions comme en Pascal.

1) Définition de fonction

Syntaxe d'une fonction :

```
[visibilité/classe] [type_de_retour] nom-de-fonction([déclarations de paramètres])
{
        Instructions
}
```

- Visibilité/Classe : Cet élément sera vu dans le chapitre traitant de la visibilité et des classes d'allocation des variables.
- Type de retour : Cet élément indique le type du résultat renvoyé par la fonction. Ce type peut être simple (int, char, etc.) ou plus complexe. Si la fonction ne renvoie aucun résultat, un type spécial "***void***" permet d'indiquer ce fait. S'il n'est pas précisé, le type d'une fonction est int par défaut.
- Nom-de-Fonction : Le nom d'une fonction peut être n'importe quel identificateur valide.
- Déclarations d'arguments : Ce sont les données venant de l'extérieur qui sont nécessaires à l'exécution de la fonction. Ils sont placés entre parenthèses. Avec ou sans argument les parenthèses sont obligatoires. La position des paramètres est primordiale.

Si la fonction reçoit des paramètres, ils seront déclarés sous la forme :

```
type <identificateur>, type <identificateur>...
```

Il est bien évident que le type de l'identificateur qui stockera la valeur du paramètre reçu par la fonction doit être le même que le type de la valeur envoyée par la fonction appelante. Si tel n'est pas le cas il y aura conversion implicite si c'est possible, avec perte si le type du paramètre reçu n'est pas suffisanmment grand pour stocker le paramètre envoyé.

Chaque paramètre est séparé d'un autre par une virgule. Ces paramètres sont locaux à la fonction où ils sont déclarés. Deux fonctions distinctes peuvent donc utiliser chacune des même noms de variables pour leurs paramètres.

- Bloc de la fonction : Celui ci commence par une accolade ouvrante et se termine par une accolade fermante (comme pour le "main()". A l'intérieur, on trouve, dans l'ordre :
  - La déclaration des variables locales.
  - Les instructions (ou blocs d'instructions) de la fonction.

2) Prototype des fonctions

Placés en tête du programme (au maximum avant le codage de la première fonction), les prototypes de fonctions sont facultatifs mais ont pour intérêt :

- de vérifier la validité (en quantité et qualité) des paramètres transmis aux fonctions
- de ne pas être obligé de coder la fonction appelée avant la fonction appelante appelante (ce qui est impossible si deux fonctions s'appellent l'une l'autre)

On écrit un prototype de fonction en écrivant

- le type de la fonction
- le nom de la fonction
- le type des paramètres entre parenthèses séparés par une virgule (le nom des paramètres n'est pas obligatoire). Si la fonction n'a pas à recevoir de paramètre, on peut mettre le mot clef "***void***" dans les parenthèses ou ne rien mettre.
- un point-virgule final **obligatoire**

Il n'y a à l'exécution **aucun contrôle ni sur le nombre ni sur le type des paramètres**. D'où l'intérêt d'utiliser un prototype systématique afin de détecter à la compilation les erreurs non décelées à l'exécution.
Exemple :

```
void fonc1();              // fonc1 ne reçoit rien et ne renvoie rien
void fonc2(void);          // fonc2 ne reçoit rien et ne renvoie rien (comme "fonc1")
void fonc3(int);           // fonc3 reçoit un "int" et ne renvoie rien
float fonc4(int, char);    // fonc4 reçoit un "int" et un "char" et renvoie un "float"
double fonc5(void);        // fonc5 ne reçoit rien et renvoie un "double"
double fonc6(double);      // fonc6 reçoit un double et renvoie un "double"
```

3) Appel d'une fonction

Une fonction est appelée par l'utilisation de son nom suivi de deux parenthèses (obligatoires) contenant éventuellement des paramètres effectifs. Ces derniers peuvent être soit des variables, soit des constantes, soit des expressions.
Un appel de fonction étant une instruction, celle-ci a pour valeur résiduelle la valeur renvoyée par la fonction et cette valeur peut être utilisée comme n'importe quelle autre valeur.
Exemple :

```
fonc1();                   // Appel de "fonc1" sans paramètre ni utiliser ce qu'elle renvoie
fonc2();                   // Appel de "fonc2" sans paramètre ni utiliser ce qu'elle renvoie
fonc3(1);                  // Appel de "fonc3" en lui passant une valeur numérique
fonc4(1 + 3, a);           // Appel de "fonc4" avec deux paramètres (un calcul valant "4" et "a")
resultat=fonc4(i, j);      // Appel de "fonc4" en récupérant son résultat dans une variable
resultat=2 * fonc5();      // Utilisation du résultat de "fonc5" dans une expression
fonc6(2 * fonc5());        // Appel de "fonc6" en lui passant un calcul utilisant "fonc5"
fonc6(fonc6(0.0));         // Appel de "fonc6" en lui passant un calcul utilisant "fonc6"
```

4) Passage de paramètres

Chaque paramètre est initialisé automatiquement avec la valeur passée par l'appelant.
A l'appel de la fonction, les valeurs passées par l'appelant (paramètres "réels" ou "effectifs") sont **recopiés** dans les identifiants de la fonction (paramètres "formels") et cette dernière travaille sur ces copies stockées dans la pile (*cf. cours sur les processus Unix*). Le nom donné à ces variables de copie importe peu.
A la sortie de la fonction, la zone de la pile occupée par les paramètres est libérée. Les valeurs de l'appelant n'ayant pas été touchées, celles-ci ne **sont jamais modifiées** par la fonction sauf si un des paramètres passés est un "tableau" (*cf. paragraphe sur les cas particuliers*). Dans ce cas, la fonction recevant un tableau reçoit l'original et non une copie et peut modifier le contenu du tableau (*cf. chapitre sur les pointeurs*).

Exemple :

```
// Prototype d'une fonction "affiche" recevant un nombre entier et ne renvoyant rien (void)
void affiche(int);

// Fonction principale du programme
main()
{
	int nb=5;			// Nombre à afficher

	affiche(nb)			// Appel de la fonction "affiche"
	printf("nb=%d\n", nb);		// Affichage de "nb" pour voir s'il a changé
}

// Codage de la fonction "affiche"
void affiche(
	int x)				// Variable où stocker le paramètre reçu
{
	// Affichage du nombre reçu (but de la fonction "affiche")
	printf("Nombre reçu : %d\n", x);

	// Modification du paramètre reçu (juste pour voir)…
	x=x + 1;
}
```

5) Retour du résultat

Une fonction peut renvoyer le résultat de son travail. Ceci sera fait grâce à l'instruction "***return (expression)***" (les parenthèses ne sont pas obligatoires).
Cette instruction **interrompt** le déroulement de la fonction et exécute le retour à la fonction appelante qui peut récupérer ou simplement utiliser le résultat renvoyé.
On peut trouver plusieurs "return" dans une fonction bien que ce soit contraire à la notion de programmation structurée ; chaque "return" n'étant traité que si certaines conditions se présentent. En ce cas, lors de l'appel et selon les conditions évaluées, la fonction sortira à l'un ou à l'autre des "return".

Une fonction ne renvoyant rien (void) n'a pas l'utilité du "return". Cependant, celui-ci peut être utilisé dans la mesure où il ne sert qu'à interrompre l'exécution de la fonction sans vouloir renvoyer de résultat. On emploie alors l'instruction "return" unique (sans parenthèse ni expression).

6) La fonction "main"

La fonction "main" est une fonction au même titre que les autres. De plus, elle renvoie elle-aussi une valeur de type "int" (par défaut "0") au processus qui invoque le programme. Enfin elle peut déclarer recevoir des paramètres (sera vu ultèrieurement) ou pas (void).

7) Exercice

Ecrire une fonction permettant de calculer la racine carrée d'un nombre.
La racine carrée d'un nombre "N" (et notée "$\sqrt{N}$") correspond à un nombre "P" tel que "P x P = N"
Elle est donnée par la limite de la suite :

- ✓ $U_0 = X$ (X nombre quelconque différent de 0)
- ✓ $U_{n+1} = ½ (U_n + N/U_n)$

```
// Fonction "racine"
double racine(
        float nb)                       //Paramètre recevant le nombre dont on veut la racine
{
        // Déclaration des variables
        double calc;                    // Calcul de la racine dans la boucle
        double diff;                    // Pour tester la différence avec la valeur précédente

        // Si la racine n'a pas besoin d'être calculée (pour "0" ou "1")
        if (nb == 0.0 || nb == 1.0)
                return (nb);            // Ici, racine de "nb" vaut "nb"

        // Initialisation début de boucle
        calc=nb;                        // Ou n'importe quel nombre différent de "0"

        do {
                // Récupération de "calc" pour comparer avec le calcul suivant
                diff=calc;

                // Approximation de la racine (le calcul se fera en précision "double")
                calc=0.5 * (calc + nb / calc);

        // Tant que le processeur peut faire la différence entre ce calcul et le précédent (limite)
        } while (calc != diff);

        // Renvoi du résultat calculé
        return calc;                    /* Ou bien "return diff" puisque ici, "diff" = "calc" */
}

// Fonction principale du programme (juste pour l'exemple)
int main(void)
{
        // Déclaration des variables
        float nb;                       // Nombre dont on veut la racine

        // Saisie et affichage de la racine
        printf("Entrez votre nombre : ");
        scanf("%f", &nb);
        printf("La racine de %f est %lf\n", nb, racine(nb));
        return 0;
}
```

Remarque : La racine carrée d'un nombre "N" peut aussi être donnée par l'algorithme suivant : choisir 2 variables "X" et "Y" avec "X" différent de "0" et boucler sur les opérations suivantes :

- ☞ Y = N / X
- ☞ X = (X + Y) / 2

A chaque boucle les variables "X" et "Y" vont se rapprocher de la racine de "N" par encadrement

8) Cas particulier : fonction recevant un tableau en paramètre

Un programmeur désirant appeler une fonction en lui passant un tableau dans l'un des paramètres appelle la fonction en mettant entre les parenthèses juste le nom du tableau sans crochet ni indice (comme pour "printf()" ou "scanf()" qui sont des fonctions pouvant traiter des chaînes de caractères).
Du coté de codage de la fonction, il faut déclarer une variable de même type que la variable reçue, c'est à dire un tableau. Quelle que soit sa dimension, on peut omettre, dans la déclaration du tableau, la longueur de la première dimension (nombre entre les premiers crochets).
Attention, une fonction recevant un tableau en paramètre a possibilité de modifier le contenu du tableau, chose impossible quand le paramètre est une variable simple (*cf. chapitre sur les pointeurs*).
Exemple :

> Ecrire une fonction permettant d'afficher une chaîne de caractères à l'écran (comme le fait "*printf*")
> Rappel : Une chaîne de caractères est un tableau de caractères dont le dernier vaut "0"

```
// Prototype fonction "affiche" recevant un tableau de caractères et renvoyant un entier long
unsigned long affiche(char[]);                  // Remarquez ici les crochets vides

// Fonction principale du programme
int main(void)
{
        // Déclaration des variables
        char chaine[100];                       // Chaîne à afficher (100 est choisi arbitrairement)

        // Saisie de la chaine
        printf("Entrez votre chaîne : ");
        scanf("%s", chaine);                    // Attention, pas de "&" pour un tableau

        // Affichage de la chaine
        printf("Votre chaîne saisie est : ");
        affiche(chaine);                        // Passage du tableau à la fonction sans mettre de crochet

        // Fin du programme
        return 0;
}

// Fonction "affiche" – Elle reçoit un tableau de caractères et renvoie un entier long non-signé
unsigned long affiche(
        char chaine[]),                         // Chaîne à afficher (la longueur "100" n'est pas obligatoire)
{
        // Déclaration des variables
        unsigned long ind;                      // Indice de boucle sur le tableau

        // Boucle sur la chaîne jusqu'à la valeur "0" conventionnelle
        for (ind=0; chaine[ind] != '\0'; ind++)
                // Affichage de chaque lettre de la chaîne
                printf("%c", chaine[ind]);

        // Il se trouve que, ici, "ind" contient la longueur réelle de la chaîne
        return ind;                     // Renvoi de la longueur (tant qu'à faire…)
}
```

9) Cas particulier : fonction devant renvoyer un tableau

Il n'est pas possible, dans l'état actuel, de faire renvoyer un tableau par une fonction, rappelons-le, un tableau n'est pas une entité manipulable par le langage.
Il faut, pour régler le problème, utiliser les pointeurs (*cf. chapitre sur les pointeurs*).

VII) LA RECURSIVITE

Il a été vu au chapitre sur les fonctions que celles-ci conservent leur environnement de travail pendant qu'une sous-fonction est exécutée et qu'il n'y a pas mélange des variables puisque chaque fonction possède une zone de travail indépendante des autres fonctions. Ceci autorise alors une fonction à s'appeler **elle-même** (fonction récursive). La fonction appelante sera alors mise en attente du résultat obtenu par la fonction appelée (qui peut elle-aussi être mise en attente par le résultat …).
Il est évident qu'une fonction récursive doit posséder une condition d'arrêt de récursivité, car sinon, la fonction s'appellerait elle-même à l'infini. La condition d'arrêt se place en général (par facilité de programmation) au début du code de la fonction mais ce n'est nullement une obligation.

1) Récursivité simple

La récursivité simple est le cas le plus courant. La fonction s'appelle elle-même une fois jusqu'à arriver au bout de sa récursivité.
Exemple : La factorielle de "n" noté" "n !"

- si n = 0, alors factorielle(n) = 1
- sinon, factorielle(n) = n x factorielle(n – 1)

```
// Fonction récursive "factorielle"
unsigned long fact (
        unsigned short n)                       // Nombre dont on veut la factorielle
{
        /* Vérification fin de récursivité */
        if (n == 0)                             // On peut aussi mettre "if (n <= 1)"
                return 1;                       // Factorielle de "0" (ou "1") vaut "1"

        // Renvoi du résultat de "n" multiplié par ce que vaut la fonction pour "n – 1"
        return n * fact(n – 1);

        // Ce que vaut "fact(n – 1)" sera d'abord calculé dans une sous-fonction etc…
}
```

2) Récursivité double

La récursivité double est le cas où la fonction s'appelle elle-même deux (ou plusieurs) fois. Le nombre d'empilements mémoire est alors phénoménal.
Exemple : La suite de Fibonacci est une suite numérique où les deux valeurs "U_0" et "U_1" sont déjà établies et la valeur "U_n" pour n >= 2 est donné par l'addition des deux termes précédents ($U_n = U_{n-2} + U_{n-1}$).

```
// Fonction récursive "Fibonacci" – On considère que U0 vaut "1" et U1 vaut "2"
unsigned long fib (
        unsigned short n)                       // Nombre dont on veut la valeur de Fibonacci
{
        // Vérification fin de récursivité
        if (n == 0)
                return 1;                       // On renvoie la valeur correspondante à U0
        if (n == 1)
                return (2);                     // On renvoie la valeur correspondante à U1

        // Renvoi du résultat de l'addition des deux appels inférieurs
        return (fib(n – 2) + fib(n – 1));
}
```

3) Récursivité croisée

La récursivité croisée est le rare cas où une fonction "A" appelle une fonction "B" qui, elle-même, appelle la fonction "A". Si un programmeur reprend le travail d'un autre, il peut avoir du mal à déceler qu'il y a récursivité d'où l'importance des commentaires.

4) Conclusion

La récursivité est un mécanisme offrant une très grande souplesse de codage au programmeur, mais se payant par une consommation très importantes des ressources du système. En effet, à chaque appel récursif, ce dernier doit sauvegarder tout le contexte de la fonction afin de pouvoir offrir à la sous-fonction une zone de travail vierge. Par exemple, pour la valeur "6", la fonction "factorielle" s'empilera 7 fois en mémoire tandis que la fonction "Fibonacci" s'empilera 24 fois. Tous ces empilements seront autant de contextes de travail différents et individuels mais gros consommateurs de ressources système lors du déroulement du programme.
Avant d'employer la récursivité, il faut donc se demander si elle ne peut pas être évitée par l'emploi de boucles appropriées.

Exemple : La factorielle de "n" sans récursivité

```
unsigned long fact (
        unsigned short n)                       // Nombre dont on veut la factorielle
{
        // Déclaration des variables
        unsigned short i;                       // Indice de boucle
        unsigned long resultat=1;               // Résultat de la factorielle

        // Boucle de factorielle
        for (i=2; i <= n; i++)                  // Si "n" vaut 0 ou 1, boucle non exécutée
                resultat=resultat * i;          // Calcul de la factorielle intermédiaire

        // Renvoi du résultat (toujours au moins égal à 1)
        return resultat;
}
```

Exemple : La suite de Fibonacci sans récursivité

```
unsigned long fib (
        unsigned short n)                       // Nombre dont on veut la valeur de Fibonacci
{
        // Déclaration des variables
        unsigned long U[3]={1, 2};              // Tableau de 3 "Un" (on n'initialise que U0 et U1)

        // Test des valeurs particulières
        if (n == 0 || n == 1)
                return (U[n]);                  // Renvoi U0 ou U1 si "n" vaut "0" ou "1"

        // Boucle de Fibonacci
        for (i=2; i <= n; i++)
        {
                U[2]=U[0] + U[1];               // Calcul de l'élément en cours
                U[0]=U[1];                      // Décalage de U1 vers U0
                U[1]=U[2];                      // Décalage de U2 vers U1
        }

        // Renvoi du résultat (dernier U2)
        return (U[2]);
}
```

5) Exercice – Changement de base

Ecrire une fonction récursive affichant n'importe quel nombre dans une base infèrieure à 10.
Rappel : Pour afficher un nombre "n" dans une base "b", on divise euclidiennement "n" par "b" tant qu'on peut diviser. Dès que le résultat de la division atteint "0", on affiche dans l'ordre inverse tous les restes des divisions précédentes.

```
// Prototype fonction "affiche" recevant un nombre long (à afficher) et un nombre court (la base)
void affiche(unsigned long, unsigned short);

// Fonction principale du programme
int main(void)
{
	// Déclaration des variables
	unsigned long nb;				// Nombre à afficher
	unsigned short base;				// Base d'affichage

	// Saisie du nombre à afficher
	printf("Entrez le nombre à afficher : ");
	scanf("%lu", &nb);

	// Saisie de la base d'affichage (contrôle de la saisie)
	base=0;					// Pour être certain que le "while" sera vrai
	while (base < 2 || base > 9)		// Tant que base < 2 ou base > 9
	{
		printf("Entrez la base d'affichage (entre 2 et 9) : ");
		scanf("%hu", &base);
	}

	// Affichage
	printf("Le nombre %lu en base %hu=", nb, base);
	affiche(nb, base);
	printf("\n");

	// Fin du programme
	return 0;
}

// Fonction "affiche"
void affiche(
	unsigned long nb				// Nombre à afficher
	unsigned short base)				// Base d'affichage
{
	// Déclaration des variables
	// Aucune variable

	// Vérification fin récursivité
	if (nb == 0)
		return;				// Fin récursivité

	// Appel récursif
	affiche(nb / base, base);

	// Affichage du reste (placé après l'appel récursif pour être affiché lors de la remontée)
	printf("%hu", nb % base);			// Le modulo "%" donne le reste d'une division
}
```

VIII) ETUDE DE CAS : LE TRIANGLE DE PASCAL – ETUDE DES DIFFERENTS SOLUTIONS

Ecrire un programme affichant le triangle de Pascal sur "n" lignes
Rappel : Le triangle de Pascal permet de donner les coefficients de multiplication pour développer l'expression $(a + b)^n$. Les coefficients de cette expression sont ceux de la ligne "n".
Chaque ligne "n" est construite de la façon suivante :
- elle contient "n + 1" chiffres
- le premier et le dernier sont "1" (sauf pour la ligne "0" où il n'y a qu'une fois le chiffre "1")
- chaque chiffre intermédiaire est l'addition du chiffre placé au dessus et du chiffre placé avant ce dernier

Exemple :
- ligne 0 : 1
- ligne 1 : 1 1
- ligne 2 : 1 2 1
- ligne 3 : 1 3 3 1
- ligne 4 : 1 4 6 4 1
- ligne 5 : 1 5 10 10 5 1
- ligne 6 : 1 6 15 20 15 6 1

etc...
Ce triangle peut aussi être calculé en utilisant les combinaisons (C) des probabilités
La ligne "n" est donné par la suite des chiffres C^0_n, C^1_n... C^n_n
La combinaison C^x_y est donné par la formule : y ! / (x ! * (y - x) !)

Le triangle de pascal a pour but de montrer le développement de $(a + b)^n$
Par exemple la ligne "5" du triangle permet de développer $(a + b)^5$
$(a + b)^5 = \mathbf{1}a^5 + \mathbf{5}a^4b^1 + \mathbf{10}a^3b^2 + \mathbf{10}a^2b^3 + \mathbf{5}a^1b^4 + \mathbf{1}b^5$

1) Principe des différentes solutions

Toutes les solutions développées dans cette étude de cas auront sensiblement le même squelette de programme

- Fonction principale du programme : cette fonction aura pour but de faire saisir le nombre de lignes voulu pour le triangle et de l'afficher
- Fonction "ligne" : cette fonction aura pour but d'afficher une ligne du triangle et sera appelée autant de fois qu'il le faut par la fonction principale.
- Fonction "comb" : cette fonction calculera la combinaison voulue. Elle sera appelée autant de fois qu'il le faut par la fonction "ligne" puisqu'une ligne du triangle de Pascal est une liste de combinaisons
- Fonctions diverses relatives aux différentes solutions permettant d'aider au calcul de la combinaison

2) Solution : Utilisation de la factorielle pour calculer les combinaisons

```
// Définitions des alias sur des types existants (plus pratique pour programmer)
typedef unsigned short ushort;                  // Entier court non signé
typedef unsigned long ulong;                    // Entier long non signé

// Prototypes des fonctions utilisées
double fact(ushort);                            // Factorielle
ulong comb(ushort, ushort);                     // Combinaison
void ligne(ushort);                             // Ligne du triangle

// Fonction principale
int main(void)
{
        // Déclaration des variables
        ushort i;                               // Indice de boucle
        ushort limite;                          // Limite du triangle demandée

        // Saisie de la limite du triangle
        printf("Entrez la limite du triangle voulue : ");
        scanf("%hu", &limite);

        // Boucle jusqu'au niveau demandé
        for (i=0; i <= limite; i++)
                // Affichage de la ligne "i"
                ligne(i);

        // Fin du programme
        return 0;
}

// Affichage d'une ligne "lig"
void ligne (
        ushort lig)                             // N°de ligne à afficher
{
        // Déclaration des variables
        ushort col;                             // Indice de colonne

        // Boucle jusqu'au niveau demandé
        for (col=0; col <= lig; col++)
                // Affichage de la combinaison C(col, lig)
                printf("%lu ", comb(col, lig));

        // Fin de ligne
        printf("\n");
}
```

```
// Calcul de la combinaison C(x, y)
ulong comb(
	ushort col,				// Colonne
	ushort lig)				// Ligne
{
	// Déclaration des variables
	// Pas de variable

	// Si on est en première ou dernière colonne (facultatif mais optimise la fonction)
	if (col == 0 || col == lig)
		return 1;			// Valeur du triangle connue

	// Si on est en seconde ou avant-dernière colonne (facultatif mais optimise la fonction)
	if (col == 1 || col == (lig - 1))
		return lig;			// Valeur du triangle connue

	// Renvoi du calcul
	return (fact(lig) / (fact(col) / fact(lig - col));
}

// Calcul de la factorielle
double fact(
	ushort nb)				// Nombre à élever en factorielle
{
	// Déclaration des variables
	// Pas de variable

	// Si "nb" vaut "0" ou "1", la factorielle vaut "1"; si "nb" vaut "2", la factorielle vaut "2")
	if (nb <= 2)
		// Renvoi de la factorielle directe
		return (nb != 0 ?nb :1);

	// Renvoi de "nb" multiplié par la valeur de la factorielle pour "(nb - 1)"
	return (nb * fact(nb - 1));
}
```

Cette solution a l'inconvénient d'appeler 3 fois la fonction "fact" pour chaque combinaison et cette dernière va générer de très grands nombres qu'il faudra stocker (d'où la raison du type "double" de cette fonction) avant que ceux-ci ne se divisent mutuellements dans la combinatoire. Ce programme est donc limité par la valeur maximale que peut donner un nombre au format "double". Le triangle devient d'ailleurs faux à partir de la 24°ligne.

3) Solution : Méthode par addition des valeurs de la ligne précédente

```
// Définitions des alias sur des types existants (plus pratique pour programmer)
typedef unsigned short ushort;                  // Entier court non signé
typedef unsigned long ulong;                    // Entier long non signé

// Prototypes des fonctions utilisées
ulong comb(ushort, ushort);                     // Combinaison
void ligne(ushort);                             // Ligne du triangle

// Fonction principale
int main(void)
{
        // Déclaration des variables
        ushort i;                               // Indice de boucle
        ushort limite;                          // Limite du triangle demandée

        // Saisie de la limite du triangle
        printf("Entrez la limite du triangle voulue : ");
        scanf("%hu", &limite);

        // Boucle jusqu'au niveau demandé
        for (i=0; i <= limite; i++)
                // Affichage de la ligne "i"
                ligne(i);

        // Fin du programme
        return 0;
}

// Affichage d'une ligne "lig"
void ligne (
        ushort lig)                             // N°de ligne à afficher
{
        // Déclaration des variables
        ushort col;                             // Indice de colonne

        // Boucle jusqu'au niveau demandé
        for (col=0; col <= lig; col++)
                // Affichage de la combinaison C(lig, col)
                printf("%lu ", comb(col, lig));

        // Fin de ligne
        printf("\n");
}
```

```
// Calcul de la combinaison C(x, y) par addition
ulong comb(
	ushort col,				// Colonne
	ushort lig)				// Ligne
{
	// Déclaration des variables
	// Pas de variable

	// Si on est en première ou dernière colonne (fin de récursivité)
	if (col == 0 || col == lig)
		return 1;			// Valeur du triangle connue

	// Si on est en seconde ou avant-dernière colonne (facultatif mais optimise la fonction)
	if (col == 1 || col == (lig - 1))
		return lig;			// Valeur du triangle connue

	// Renvoi de l'addition des deux nombres du dessus
	return (comb(col - 1, lig - 1) + comb(col, lig - 1));
}
```

Cette solution semble plus intéressante car on élimine d'abord la factorielle génératrice de grands nombres. De plus, en première lecture, on voit qu'il n'y a plus que 2 appels récursifs au lieu des 3 de la solution précédente. Cependant si on regarde mieux on s'aperçoit qu'on est dans un cas de double récursivité où chaque appel récursif en enchaîne 2 autres ce qui amène rapidement à saturation du système.
Effectivement cette solution convient un peu mieux que la précédente car elle ne génère pas de grands nombres et elle fonctionne pour un triangle de plus de 25 lignes mais le temps de traitement s'accroît exponentiellement en fonction de la ligne affichée.

4) Solution : Conservation des combinaisons déjà calculées

```
// Définitions des alias sur des types existants (plus pratique pour programmer)
typedef unsigned short ushort;                  // Entier court non signé
typedef unsigned long ulong;                    // Entier long non signé

// Prototypes des fonctions utilisées
void comb(ushort, ushort, ulong[501][501]);     // Combinaison
void ligne(ushort, ulong[501][501]);            // Ligne du triangle

// Fonction principale
int main(void)
{
        // Déclaration des variables
        ushort i;                               // Indice de boucle
        ushort limite;                          // Limite du triangle demandée
        ulong tab_comb[501][501];               // Tableau des combinaisons calculées

        // Saisie et contrôle de la limite du triangle
        while (1)
        {
                printf("Entrez la limite du triangle voulue (max. 500) : ");
                scanf("%hu", &limite);
                if (limite <= 500)
                        break;
                printf("%hu trop grand – Recommencez\n", limite);
        }

        // Boucle jusqu'au niveau demandé
        for (i=0; i <= limite; i++)
                // Affichage de la ligne "i" et mémorisation dans le tableau des combinaisons*/
                ligne(i, tab_comb);

        // Fin du programme
        return 0;
}

// Affichage d'une ligne "lig" et mémorisation de cette ligne dans le tableau des combinaisons
void ligne (
        ushort lig,                             // N°de ligne à afficher
        ulong tab_comb[501][501])               // Tableau des combinaisons
{
        // Déclaration des variables
        ushort col;                             // Indice de colonne

        // Boucle jusqu'au niveau demandé
        for (col=0; col <= lig; col++)
        {
                // Mémorisation de la combinaison dans le tableau
                comb(col, lig, tab_comb);

                // Affichage de la combinaison C(col, lig)
                printf("%lu ", tab_comb[lig][col]);
        }

        // Fin de ligne
        printf("\n");
}
```

```
// Mémorisation de la combinaison C(x, y) dans le tableau
void comb(
        ushort col,                             // Colonne
        ushort lig,                             // Ligne
        ulong tab_comb[501][501])               // Tableau des combinaisons
{
        // Déclaration des variables
        // Pas de variable

        // Si on est en première ou dernière colonne (pas assez d'antécédents))
        if (col == 0 || col == lig)
        {
                tab_comb[lig][col]=1;           // Valeur du triangle connue
                return;                         // Plus la peine de continuer
        }

        // Si on est en seconde ou avant-dernière colonne (facultatif mais optimise la fonction)
        {
                tab_comb[lig][col]=lig;         // Valeur du triangle connue
                return;                         // Plus la peine de continuer
        }

        // Ici, on est certain de ne pas être dans les valeurs extrèmes du triangle*/

        // Mémorisation de l'addition des deux nombres du dessus
        tab_comb[lig][col]=tab_comb[lig – 1][col – 1] + tab_comb[lig – 1][col];
}
```

Cette solution évite la récursivité mais implique la gestion d'un tableau de valeurs calculées. Elle implique aussi de connaître à l'avance le nombre de lignes du triangle (donc oblige à programmer un contrôle sur la saisie) et un gaspillage de 50% de la mémoire car on est obligé de déclarer un tableau carré alors que les valeurs calculées n'en occupent que sa moitié. Cependant la rapidité d'exécution est exponentiellement plus avantageuse que toutes les solutions précédentes et permet un triangle de 500 lignes tout en étant facilement modifiable pour calculer des triangles plus gros. Et enfin l'utilisation de fonctions d'allocation dynamique de mémoire permettra d'éliminer le gaspillage de mémoire en n'allouant que ce qui est nécessaire au stockage mais nécessitera une programmation plus poussée (*cf. chapitre sur la gestion de la mémoire*).

5) Solution : Calcul de la combinaison par simplification de la fraction

```
// Définitions des alias sur des types existants (plus pratique pour programmer)
typedef unsigned short ushort;                  // Entier court non signé
typedef unsigned long ulong;                    // Entier long non signé

// Prototypes des fonctions utilisées
ulong comb(ushort, ushort);                     // Combinaison
void ligne(ushort);                             // Ligne du triangle

// Fonction principale
int main(void)
{
        // Déclaration des variables
        ushort i;                               // Indice de boucle
        ushort limite;                          // Limite du triangle demandée

        // Saisie et contrôle de la limite du triangle
        printf("Entrez la limite du triangle voulue : ");
        scanf("%hu", &limite);

        // Boucle jusqu'au niveau demandé
        for (i=0; i <= limite; i++)
                // Affichage de la ligne "i"
                ligne(i);

        // Fin du programme
        return 0;
}

// Affichage d'une ligne "lig"
void ligne (
        ushort lig)                             // N°de ligne à afficher
{
        // Déclaration des variables
        ushort col;                             // Indice de colonne

        // Boucle jusqu'au niveau demandé
        for (col=0; col <= lig; col++)
                // Affichage de la combinaison C(lig, col)
                printf("%lu ", comb(col, lig));

        // Fin de ligne
        printf("\n");
}
```

```
// Calcul de la combinaison C(x, y) par simplification de la fraction
ulong comb(
    ushort col,                                 // Colonne
    ushort lig)                                 // Ligne
{
    // Déclaration des variables
    ushort i;                                   // Indice de boucle
    ushort j;                                   // Indice de boucle
    ushort pivot;                               // Pivot de départ de la multiplication
    ulong res;                                  // Résultat de la combinaison

    // Si on est en première ou dernière colonne (facultatif mais optimise la fonction)
    if (col == 0 || col == lig)
        return 1;                               // Valeur du triangle connue

    // Si on est en seconde ou avant-dernière colonne (facultatif mais optimise la fonction)
    if (col == 1 || col == (lig - 1))
        return lig;                             // Valeur du triangle connue

    // Recherche du pivot (point à partir duquel on commencera à multiplier)
    pivot=(lig - col) > col ?(lig - col) :col;

    // Calcul de la combinaison par multiplication et division simultanée
    res=1;
    for (i=pivot + 1, j=1; i <= lig; i++, j++)
        res=res * i / j;

    // Renvoi du résultat
    return (res);
}
```

Cette solution peut-être considérée comme la meilleure. Elle bénéficie du fait que le calcul de C^3_8 tout comme le calcul de C^5_8 sont identiques et se font en faisant 6 x 7 x 8 / 1 x 2 x 3 (les nombres "3" et "8 – 3" ou "5" et "8 – 5" sont des nombres pivots de ce calcul).
De plus, grace à l'évolution parallèle de deux indices, on peut multiplier et diviser simultanément ce qui évite au résultat de la combinaison de grossir démesurément et inutilement. Et mathématiquement, la division sera toujours parfaite puisque le second indice commence à "1".

6) Conclusion

Différentes solutions même programmées avec rigueur ne produisent pas forcément un travail réellement efficace car elles sont limitées d'une part par les contraintes du langage, et d'autre part par les limites du système sur lequel sera implanté le programme.
Il vaut mieux réfléchir d'abord au problème et à l'optimisation éventuelle de sa solution avant de commencer à coder.

IX) LA VISIBILITE DES VARIABLES - LES CLASSES D'ALLOCATION DES VARIABLES

1) Visibilité des variables

La visibilité d'une variable concerne l'endroit du source où la variable est accessible (lisible ou modifiable).

a) Variable locale

Une variable locale est une variable qui a été déclarée dans un bloc d'instructions. Rappel : Chaque bloc d'instructions (encadré par des accolades **{}**) peut se composer de variables (déclarées avant toute instruction) ; et d'instructions.
La variable locale est locale pour le bloc et tous ses sous-blocs éventuels. Elle ne sera donc accessible (lisible ou modifiable) que depuis le bloc où elle a été déclarée et tous ses sous-blocs éventuels.
Puisque la variable n'est vue que du bloc, rien n'empêche deux blocs distincts d'avoir une variable de même nom. Si des blocs imbriqués ont chacun une variable du même nom, le sous-bloc ne verra plus que sa variable et ne verra plus la variable déclarée dans le bloc supérieur.
Une variable locale qui n'est pas initialisée par le programmeur est créée par défaut avec un contenu imprévisible.
Exemple :

```
int main(void)
{                                       // Bloc numéro 1
        int var;                        // Création variable "var" du bloc "1"

        var=10;                         // Affectation variable "var" du bloc "1"

        {                               // Bloc numéro 1.1
                var=50;                 // Affectation variable "var" du bloc "1"
                printf("var=%d\n", var);// Affichera "50"
        }                               // Fin de bloc numéro 1.1

        printf("var=%d\n", var);        // Affichera encore "50"

        {                               // Bloc numéro 1.2
                int var;                // Création variable "var" du bloc "1.2"

                var=100;                // Affectation variable "var" du bloc "1.2"
                printf("var=%d\n", var);// Affichera "100"
        }                               // Fin de bloc numéro 1.2

        printf("var=%d\n", var);        // Affichera toujours "50" ("bloc "1.2" terminé)
}
```

b) Variable globale

Une variable globale est une variable qui a été déclarée en dehors de tout bloc d'instructions. Elle sera alors visible depuis sa déclaration jusqu'à la fin du fichier source mais sera masquée par toute variable locale du même nom.
Par ailleurs, une variable globale sera connue de tout le programme. On pourra donc y accéder dans une fonction même située dans un autre source à condition de la déclarer en "*extern*" dans ce dernier (*cf. chapitre sur les classes d'allocation*). Cependant, les programmeurs n'aiment pas utiliser des variables globales car en cas d'incohérence du contenu de la variable dans un programme mal écrit, il est difficile de repérer l'endroit où a pu se produire l'erreur d'écriture.
Rien n'empêche de déclarer dans un bloc quelconque une variable locale de même nom qu'une variable globale. A partir de ce moment et jusqu'à la fin du bloc; la variable globale sera masquée par la variable locale et le langage n'aura plus accès qu'à cette dernière.
Une variable globale qui n'est pas initialisée par le programmeur est créée par défaut avec son contenu mis à zéro.
Exemple :

```
// Prototypes de trois fonctions neutres
void fonction1(void);
void fonction2(void);
void fonction3(void);

int main(void)
{
	int var;				// Création variable locale "var"

	var=10;
	printf("var=%d\n", var);		// Affichera "10"
	fonction1();				// Appel de "fonction1"
}

int var=50;					// Variable globale mais pas connue du "main"

// Fonction "fonction1"
void fonction1(void)
{
	printf("var=%d\n", var);		// Affichera "50" (la variable globale)
	var=100;				// Modification de la variable globale
	fonction2();				// Appel de "fonction2"
}

// Fonction "fonction2"
void fonction2(void)
{
	int var;				// Création variable locale "var"

	var=200;
	printf("var=%d\n", var);		// Affichera "200" (la variable locale)
	fonction3();				// Appel de "fonction3"
}

// Fonction "fonction3"
void fonction3(void)
{
	printf("var=%d\n", var);		// Affichera "100" (la variable globale)
}
```

2) Classes d'allocation des variables

La classe d'allocation d'une variable concerne la façon dont sera stocké la variable dans la mémoire ; ce qui influe sur sa durée de vie (temps d'exécution du programme durant lequel on est certain de conserver le contenu de la variable intacte).

a) Classe automatique (mot-clef "auto")

La variable est créée et supprimée de la mémoire automatiquement par le programme quand il n'en a plus besoin. C'est le cas entre autres des paramètres des fonctions. Toute variable définie sans classe d'allocation est placée en classe automatique par défaut. Toutes les variables vues jusqu'à présent dans les exemples étaient en classe automatique.

```
auto int n=0;                // Le mot clef "auto" n'est pas obligatoire
```

b) Classe registre (mot-clef "register")

La variable est stockée dans un registre (mémoire très rapide) si c'est possible (registre disponible, taille de la variable suffisamment petite pour y tenir, etc.). Dans le cas contraire, le compilateur ne signale rien mais considère la variable comme automatique. On peut optimiser les boucles d'un programme en plaçant les indices de boucle dans des registres mais aujourd'hui, les compilateurs sont très performants et optimisent les variables d'un programme mieux qu'un programmeur ne sait le faire. Il vaut donc mieux ne pas interférer avec le compilateur en n'essayant pas de placer des variables dans des registres à la place de ceux-ci. Par ailleurs, il ne faut pas espérer pouvoir placer un double (8 octets), un tableau ou une chaîne de caractères (tableau de caractères) dans un registre.

```
register int n=0;            // Si la mise en registre n'est pas possible, "n" sera "auto"
```

c) Classe statique (mot-clef "static")

La variable est stockée dans une zone mémoire et y restera jusqu'à la fin d'exécution du programme. Cela garanti donc, dans le cas de fonction appelée plusieurs fois, que la variable n'est pas perdue entre chaque appel (permet de programmer un compteur d'appels par exemple).
Par ailleurs, dans le cas d'un programme écrit avec plusieurs sources, la variable n'est connue que du source dans lequel elle est définie. Ceci permet ainsi d'avoir plusieurs variables globales ou plusieurs fonctions portant un même nom mais n'ayant pas un même but ; chaque variable globale ou chaque fonction à but différent étant placé dans un source différent.
Enfin, cela permet d'avoir une variable dont la valeur initiale est garantie car elle est initialisée par défaut à 0. De plus, si elle est initialisée lors de la déclaration, cette initialisation ne se fait qu'une seule fois ; même si la fonction où est déclarée cette variable est appelée plusieurs fois.

```
static int n=100;            // L'initialisation à "100" ne se fera qu'une seule fois
```

Exemple :

```
// Fonction quelconque
void fonction(void)
{
    // Déclaration des variables
    static int n=0;          // Variable statique (valeur conservée) initialisée une fois

    // Programmation
    n++;                     // Valeur incrémentée à chaque appel de "fonction()"
…
```

d) Classe externe (mot-clef "extern")

Cette classe permet de déclarer une variable sans lui réserver de place en mémoire. Cela est en général utilisé dans le cas d'un programme tenant sur plusieurs fichiers sources où la même variable (forcément globale) est utilisée dans les différents fichiers sources.
En effet, pour passer du "*.c*" au "*.o*", le compilateur effectue une première analyse du source et toute variable utilisée dans une fonction doit être déclarée avant son utilisation. Le programmeur est donc obligé de déclarer sa même variable dans chaque fichier source. Mais si, pour chaque fichier "*.o*" généré le compilateur réserve un emplacement mémoire pour la variable déclarée, l'éditeur de lien ne comprendra pas que toutes ces variables déclarées ne concernent qu'une seule entité mémoire pour le programme et produira alors une erreur de compilation.
C'est pourquoi, la variable ne doit être définie (réservée en mémoire) que dans un seul source (en général celui qui contient la fonction "*main*") et seulement déclarée (avec le mot "*extern*") dans les autres sources. Il n'y a dans ce dernier cas aucune réservation mémoire lors de la déclaration.
C'est ici qu'on fait la différence entre "déclaration" et "définition"
Enfin il n'est pas possible de déclarer une variable en "extern" et l'initialiser en même temps car il n'y a pas de mémoire associée à la variable.

```
extern int var;                    // Interdit d'écrire "extern int var=100"
```

Exemple de travail sur la même variable avec plusieurs sources :

Source 1 (contenant la fonction "main") :

```
// Définition de "var"
int var=100;

// Déclaration de fonction "fctX"
void fctX(void);

// Programme principal
int main(void)
{
        // Appel de "fctX"
        fctX();
}
```

Source 2 (contenant la fonction "fctX") :

```
// Déclaration de "var"
extern int var;

// Définition de fonction "fctX"
void fctX(void)
{
        // Utilisation de "var"
        printf("var=%d\n", var);
}
```

Il est aussi possible de déclarer une variable en "extern" dans le même fichier source que celui ou elle est définie lorsqu'on veut, dans un soucis de rigueur de programmation, déclarer dans chaque fonction toutes les variables qu'elle utilise, y compris les variables globales. Il est donc nécessaire, pour ces dernières, de les déclarer dans la fonction en "extern".

```
int var;                           // Variable globale connue de toutes les fonctions

// Fonction quelconque
void fonction(void)
{
        // Déclaration des variables
        extern int var;

        // Ce programmeur rigoureux indique ainsi qu'il a l'intention d'utiliser la variable globale "var"
        // Un autre programmeur qui arrive dans le projet voit facilement que la fonction utilisera "var"
        // Si "var" est modifiée par erreur, il sera facile de répertorier toutes les fonctions l'utilisant
 …
```

3) Résumé visibilité/durée de vie

En sachant que "visibilité" signifie "où on peut voir la variable" et "durée de vie" signifie "combien de temps la variable garde-t-elle sa valeur, on peut établir le tableau résumé suivant :

<table>
<tr><th colspan="2"></th><th colspan="2">auto</th><th>register</th><th>static</th><th>extern</th></tr>
<tr><td rowspan="2">Variable locale</td><td>Visibilité</td><td colspan="2" rowspan="2">Le bloc dans lequel elle est définie</td><td>Le bloc dans lequel elle est définie</td><td>Le bloc dans lequel elle est déclarée</td></tr>
<tr><td>Durée de vie</td><td colspan="2">Le temps du programme</td></tr>
<tr><td rowspan="2">Variable globale</td><td>Visibilité</td><td colspan="2">Le source dans lequel elle est définie</td><td>Le source dans lequel elle est définie</td><td>Le source dans lequel elle est déclarée</td></tr>
<tr><td>Durée de vie</td><td colspan="4">Le temps du programme</td></tr>
</table>

4) Les exceptions "const" et "volatile"

Il est possible de demander au compilateur de protéger la valeur initialisée d'une variable contre toute modification ultèrieure (affectation, incrément, etc.) en mettant le mot "***const***" devant la déclaration de la variable. Le mot "***volatile***" n'existe que pour être l'opposé de "***const***" en indiquant que la variable est modifiable (toute variable est modifiable par défaut).
Il peut paraître bizarre de vouloir une variable non-modifiable. En général, on s'en sert pour protéger les tableaux passés en paramètres d'une fonction et on va indiquer ainsi que la fonction n'a pas le droit de modifier le contenu du tableau qu'elle reçoit.
Exemple :

```
void fonc(
        const int tab[])                    // Le contenu de "tab" est invariable
{
        const float pi=3.14159267;          // Le nombre "pi" est invariable

        tab[x]=…                            // Interdit quel que soit "x"
        pi=…                                // Interdit aussi
…
```

X) LES POINTEURS

Au contraire des bruits qui courent à propos du langage C, les pointeurs ne présentent aucune difficulté à appréhender pour peu qu'ils soient vus sans précipitation ni inquiétude subjective.

1) Généralités

Il a été vu que toute variable en langage C ne contient que du nombre (valeur numérique, code ascii, etc.). Cependant, chaque variable est située à un emplacement mémoire numéroté appelé aussi "adresse".
Le langage C offre au programmeur le moyen de récupérer cette adresse et de la stocker dans une autre variable.
A partir de cette seconde variable, il est naturellement possible de connaître sa valeur (comme toute variable habituelle du langage C) mais comme cette valeur correspond à l'adresse d'une première variable, il est aussi possible de connaître la valeur de la première.
Cette seconde variable qui contient l'adresse de la première est appelée "pointeur".

Dans tout ce chapitre ; et afin de distinguer les valeurs "adresse" des valeurs "classiques" (toutes ces valeurs ne sont à la base que des nombres) ; les adresses commenceront toujours par "0x" ce qui correspond à la notation numérique en hexadécimal.

2) Premier essai de pointeur

a) Premier pointeur – Le pointeur simple

Déclarons une variable appelée "*var*" et contenant la valeur "*3.1416*".

```
float var=3.1416;
```

Il se trouve que cette variable "*var*" a été stockée en mémoire à un emplacement appelé "*adresse*". Nous pouvons émettre l'hypothèse que "*var*" est stockée en mémoire à l'adresse "*0x10*".
Voici un schéma de la mémoire à ce moment là :

Adresse	Nom	Contenu
0x10	var	3.1416

Si on demande au langage "**affiche var**", il nous donnera naturellement "**3.1416**".

```
printf("La valeur de var est %f\n", var);                    // Affichera "3.1416"
```

Cependant, le langage C offre au programmeur un opérateur permettant de récupérer non plus la valeur d'une variable mais l'adresse d'une variable. L'opérateur qui permet d'avoir "*l'adresse de*" est l'opérateur unaire "***&***" dont on a déjà parlé lors de "*scan()f*".

Si on demande au langage "**affiche adresse de var**", il donnera "**0x10**".

```
printf("L'adresse de var est 0x%x\n", &var);                 // Affichera "0x10"
```

Déclarons maintenant une seconde variable appelée "*pt*" et contenant la valeur "*0x10*".

```
int pt=0x10;
```

Cette seconde variable "*pt*" a été stockée à un second emplacement mémoire (seconde adresse) de valeur "*0x20*" (par exemple).
Voici un schéma de la mémoire à ce moment là :

Adresse	Nom	Contenu
0x10	var	3.1416
0x20	pt	0x10

Si on demande au langage "**affiche pt**", il nous donnera naturellement son contenu "**0x10**".
Si on demande au langage "**affiche adresse de pt**", il donnera "**0x20**"

```
printf("La valeur de pt est 0x%x\n", pt);                    // Affichera "0x10"
printf("L'adresse de pt est 0x%x\n", &pt);                   // Affichera "0x20"
```

Cependant, ici, il est possible de demander "**affiche ce qu'il y a à l'adresse 0x10**". Et, en tout état de cause, on cherche à obtenir la valeur "**3.1416**".
Il se trouve que la valeur "**0x10**" est stockée dans "**pt**" et que l'opérateur qui permet d'avoir "*ce qu'il y a à l'adresse de*" est l'opérateur unaire "*" qui se dit aussi "*valeur pointée par*".

```
printf("La valeur de *pt est %f\n", *pt);                    // On cherche à avoir "3.1416"
```

Le soucis du langage C est que la valeur "**3.1416**" est stockée dans un **float** qui utilise 4 octets en mémoire. Lorsqu'on demande au langage le contenu de "**var**", le compilateur connaît le type de "**var**" qui est "**float**" et sait qu'il faut lire alors 4 zones mémoires contiguës pour avoir la valeur demandée (petite parenthèse, "pt" est une variable "int" donc stockée elle aussi sur 4 octets mémoires).
Voici un schéma exact de la mémoire à ce moment là :

Adresse	Nom	Contenu
0x10 0x11 0x12 0x13	var	3.1416
0x20 0x21 0x22 0x23	pt	0x10

Ainsi, quand on demande le contenu de "***pt**" (contenu de ce qu'il y a à l'adresse représentée par "pt"), le compilateur ne sait pas combien de zones mémoires il faut lire à l'adresse "**0x10**" (il peut y avoir un "char", un "double" ou un autre type à cette adresse). Bien entendu, le compilateur sait faire le lien entre "var" et son type ("float=>4 octets") mais ne sait pas faire ce lien dont on a besoin ("à l'adresse *0x10* il y a un *float*").
Pour que le compilateur puisse faire ce dernier lien, il faut lui indiquer que "**ce qui est pointé par pt** ou plus simplement "***pt**" est un "**float**" (attention, "*pt" de lit "étoile pt").

```
~~int pt=0x10;~~                                    // Cette instruction n'est pas correcte
float *pt=0x10;                                     // Voici ce qu'il faut écrire
/* Peut se lire indifféremment de deux manières différentes :
        - "étoile pt est de type float"
        - "pt est de type "float étoile" ou "pointeur sur float"
*/

printf("La valeur de *pt est %f\n", *pt);           // On aura maintenant "3.1416"
```

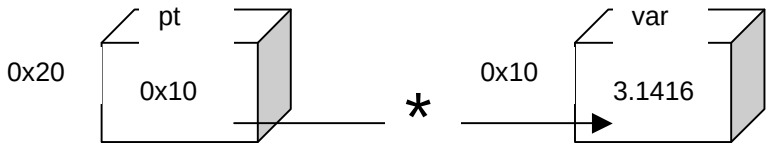
Dans la seconde et correcte déclaration, il est automatiquement déclaré deux entités utilisables par le langage :

- "*pt" est de type "float" donc le langage saura qu'il faut récupérer 4 octets mémoire si on lui demande la valeur de "*pt"
- "pt" est de type "int" contenant l'adresse d'un "float" => "pt" est de type "float étoile" (**n'oubliez jamais de prononcer le mot "étoile" quand vous parlez de votre pointeur**).

Puisque "&var" donne aussi "0x10", on peut utiliser cette dernière égalité :

```
float *pt=&var;                                     // Déclaration réelle du pointeur "pt"
// Attention, la valeur "&var" sera affectée à "pt" et non à "étoile pt" !!!
```

"pt" est un "pointeur sur un float"



b) Second pointeur – Le pointeur sur un pointeur ou double pointeur

En continuant les essais, déclarons une troisième variable appelée "*dpt*" et contenant la valeur "*0x20*".

```
int dpt=0x20;
```

Cette troisième variable "*dpt*" a été stockée à un troisième emplacement mémoire (troisième adresse) de valeur "*0x30*" par exemple.
Voici de nouveau le schéma exact de la mémoire à ce moment là :

Adresse	Nom	Contenu
0x10 0x11 0x12 0x13	var	3.1416
0x20 0x21 0x22 0x23	pt	0x10
0x30 0x31 0x32 0x33	dpt	0x20

Si on demande au langage "**affiche dpt**", il nous donnera, comme pour toute autre variable , le contenu de "*dpt*" soit "**0x20**".
Si on demande au langage "**affiche adresse de dpt**", il donnera "**0x30**"

```
printf("La valeur de dpt est 0x%x\n", dpt);          // Affichera "0x20"
printf("L'adresse de dpt est 0x%x\n", &dpt);         // Affichera "0x30"
```

Mais c'est encore le même problème, si on demande "**affiche *dpt**" (affiche "ce qui est pointé par dpt"), le compilateur ne sait pas combien de zones mémoires il faut lire à l'adresse "**0x20**".
Pour que le compilateur s'en sorte, on peut lui indiquer que "***dpt**" est un **"int**" (puisqu'à l'adresse "0x20" il y a un "int") et essayer l'écriture suivante :

```
~~int dpt=0x20~~;                                    // Cette instruction n'est pas correcte
int *dpt=0x20;                                       // Essayons celle-là

printf("La valeur de *dpt est 0x%x\n", *dpt);        // On aura probablement "0x10"
```

Si on demande maintenant "**affiche *dpt**", on aura probablement "**0x10**" et on peut se dire qu'on a réussi. Mais si on demande "**affiche **dpt**" (affiche "ce qui est pointé par ce qui est pointé par dpt"), le compilateur ne connaissant pas le type de "**dpt" est de nouveau bloqué. Pourtant, ce lien entre "***dpt*" et "*3.1416*" existe virtuellement; il ne reste plus qu'à l'indiquer au compilateur !
Pour que ce dernier s'en sorte, il faut indiquer que "****dpt**" est un "**float**".

```
~~int dpt=0x20~~;                                    // Cette instruction n'est pas correcte
~~int *dpt=0x20~~;                                   // Celle-là non plus
float **dpt=0x20;                                    // Voici ce qu'il faut écrire

printf("La valeur de *dpt est 0x%x\n", *dpt);        // On a toujours "0x10"
printf("La valeur de **dpt est %f\n", **dpt);        // Et ici on a "3.1416"
```

Dans la troisième et correcte déclaration, il est déclaré alors trois entités utilisables par le langage :

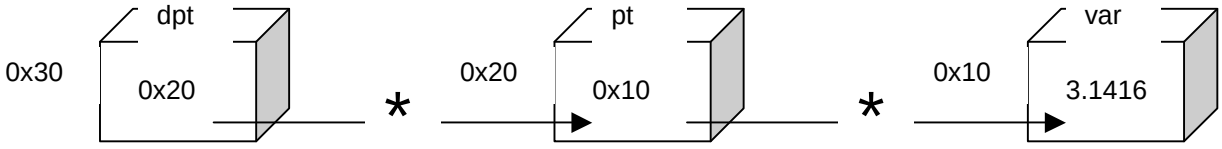
- "**dpt" est de type "float" donc le langage saura qu'il faut récupérer 4 octets mémoire si on lui demande le contenu de "**dpt"
- "*dpt" est de type "int" contenant l'adresse d'un "float" => "*dpt" est de type "float étoile" (toujours ne pas oublier le mot "étoile"). Le langage saura donc accéder au contenu de "*dpt".
- "dpt" est de type "int" contenant l'adresse de l'adresse d'un "float" => "dpt" est de type "float étoile étoile" (**en prononçant autant de fois le mot "étoile" qu'il le faut**).

Puisque "&pt" donne "0x20", on peut encore utiliser cette dernière écriture :

```
float **dpt=&pt;                                    // Déclaration réelle du pointeur "dpt"
// Toujours pareil, la valeur "&pt" sera affectée à "dpt" et non à "**dpt" !!!
```

"dpt" est un "pointeur sur un pointeur sur un float"

Remarque : Ne confondez pas "&pt" (adresse de "pt" = "0x20"), "pt" (valeur de "pt" = "0x10") et "*pt" (valeur de la case située à l'adresse "0x10" = "3.1416")



Récapitulatif
Voici un récapitulatif du premier programme manipulant des pointeurs

```
int main(void)
{
	float var=3.1416;				// Déclaration de la variable "var"
	float *pt=&var;					// Déclaration du pointeur "pt"
	float **dpt=&pt;				// Déclaration du pointeur "dpt"
	float pi;					// Déclaration d'une variable "pi"

	printf("La valeur de var est %f\n", var);	// Affichera "3.1416"
	printf("L'adresse de var est 0x%x\n", &var);	// Affichera "0x10"

	printf("La valeur de pt est 0x%x\n", pt);	// Affichera "0x10"
	printf("L'adresse de pt est 0x%x\n", &pt);	// Affichera "0x20"
	printf("La valeur de *pt est %f\n", *pt);	// Affichera "3.1416"

	printf("La valeur de dpt est 0x%x\n", dpt);	// Affichera "0x20"
	printf("L'adresse de dpt est 0x%x\n", &dpt);	// Affichera "0x30"
	printf("La valeur de *dpt est 0x%x\n", *dpt);	// Affichera "0x10"
	printf("La valeur de **dpt est %f\n", **dpt);	// Affichera "3.1416"

	// Je peux manipuler n'importe quelle des trois premières variables
	pi=var;						// Je peux utiliser "var"
	pi=(*pt);					// Je peux utiliser "*pt"
	pi=(**dpt);					// Je peux utiliser "**pt"

	// Les parenthèses sont obligatoires sinon le compilateur traduit par "pi = pi * pt"
}
```

Remarques :

☞ Un pointeur contenant toujours une adresse implique qu'un pointeur est toujours une variable au format entier quel que soit l'objet qu'il y a au bout.

☞ Si "pt" vaut "0x10", ne confondez pas "pt" => contenu de la case nommée "pt" ("0x10") et "*pt" => contenu de la case dont l'adresse est "0x10"

☞ Si on demande "*(&var)", le compilateur ira chercher "ce qui est pointé par l'adresse &var", autrement dit "var".

Les opérateurs "&" et "*" s'annulent mutuellement

3) Pointeurs et tableaux

Déclarons un tableau de 3 nombres à virgule flottante

```
float tab[3]={3.1416, 2.7183, 1.4142};
```

Voici un schéma simplifié de la mémoire à ce moment là :

Adresse	Nom	Contenu
0x1000	tab[0]	3.1416
0x1001	tab[1]	2.7183
0x1002	tab[2]	1.4142

Remarque : En fait, les adresses ne sont pas réellement celles-là car pour chaque élément, il lui est réservé 4 octets en mémoire, donc les adresses réelles sont décalées de 4 et non de 1. Mais il est préférable, pour l'instant, de les symboliser ainsi pour bien montrer que les éléments se suivent en mémoire.

Il se trouve que, lorsque le compilateur crée un tableau, il crée en plus une variable supplémentaire dont l'identificateur est le **nom du tableau sans crochet**, et le contenu est **l'adresse du premier élément** ("&tab[0]").

Adresse	Nom	Contenu
0x10	tab	0x1000

"tab" contient une adresse et au bout de cette adresse ("**tab*") ; il y a "*3.1416*" (un nombre de type "float") => "tab" est donc un "pointeur sur un float" => "tab" est donc un "float étoile" !!!

Le nom d'un tableau est toujours un pointeur sur son premier élément
tab ⇔ &tab[0]

4) Opération sur les pointeurs

Si on demande au langage "**affiche tab**", il nous donnera comme pour toute variable déjà vue le contenu de "tab", soit "**0x1000**".
Si on demande au langage "**affiche ce qui est pointé par tab**" (affiche "**tab*"), il donnera "**3.1416**" qui correspond aussi à "**tab[0]**".
Si on demande au langage "**affiche (tab + 1)**", il nous donnera naturellement le contenu de "tab" ajouté de "1" ; c'est à dire "**0x1001**"
Si on demande au langage "**affiche ce qui est pointé par (tab + 1)**" (affiche "**(tab + 1)*"), il donnera "**2.7183**" qui correspond aussi à "**tab[1]**".
Il en résulte la règle suivante

tab[n] ⇔ *(tab + n)

Enfin, si on demande l'adresse de tab[2] (*&tab[2]*), on aura "**0x1002**" qui peut aussi être obtenu par "**tab + 2**".
Il en résulte la seconde règle suivante :

&tab[n] ⇔ tab + n

Remarque : Puisque les opérateurs "**&**" et "*****" s'annulent l'un l'autre, on peut passer de la première règle à la seconde en rajoutant l'opérateur "**&**" de chaque coté de l'égalité ; et on peut passer de la seconde à la première en rajoutant l'opérateur "*****" de chaque coté de l'égalité.

Ces règles étant normalisées, elles restent valable même si on considère les adresses réelles de la mémoire (décalées de 4 dans le cas du "float" et non de 1) :

Adresse	Nom	Contenu
0x10	tab	0x1000
0x1000	tab[0]	3.1416
0x1004	tab[1]	2.7183
0x1008	tab[2]	1.4142

Ainsi, lorsqu'on rajoute "n" à un pointeur sur un type particulier, le langage rétablit le vrai calcul et multiplie "n" par la taille du type. Donc si on demande au langage "tab + 2", il nous donnera "0x1008" et non "0x1002".

"tab + n" est traduit par "tab + n * sizeof(<type de tab>)"

Ceci permet, en utilisant un second pointeur, d'accéder aux éléments d'un tableau plus rapidement qu'en utilisant les crochets "[]" car dans ce dernier cas, le compilateur insère dans le code assembleur des instructions de calcul pour accéder à l'indice "n" à partir du début du tableau alors qu'avec un pointeur, le point d'accès à l'indice "n" est déjà créé et prêt à l'emploi.

Exemple d'accès aux valeurs d'un tableau :

```
int main(void)
{
	// Déclaration des variables
	float tab[3]={3.1416f, 2.7183f, 1.4142f};		// Tableau de trois flottants
	float *pt;					// Pointeur sur un flottant
	int i;						// Indice de boucle

	// Balayage du tableau en utilisant un indice classique
	for (i=0; i < 3; i++)
		printf("tab[%u]=%f\n, i ,tab[i]);		// L'accès à tab[i] implique un calcul

	// Balayage du tableau en utilisant un pointeur (plus rapide)
	for (i=0, pt=tab; i < 3; i++, pt++)			// "i" sert seulement pour quitter la boucle
		printf("tab[%u]=%f\n, i, *pt);			// L'accès à *pt ne demande aucun calcul

	// Fin du programme
	return 0;
}
```

5) Où la logique démontre l'impossible

Prenons la première des règles démontrées précédemment : "tab[n] = *(tab + n)" et développons là jusqu'à l'extrême…

- tab[n] ⇔ *(tab + n) (première des règles)
- tab + n ⇔ n + tab (commutativité de l'addition)
- *(tab + n) ⇔ *(n + tab) par déduction
- *(n + tab) ⇔ n[tab] par simple application de la première règle

Il en résulte qu'on est autorisé à écrire "tab[n]" ou "n[tab]" en langage C lorsqu'on désire atteindre un des éléments d'un tableau

De la même façon, si on applique la première règle à un tableau à deux dimensions, on obtient le raisonnement suivant :

- tab[x][y] ⇔ *(tab + x)[y] par application simple de la première règle sur "tab[x]"
- *(tab + x)[y] ⇔ *(*(tab + x) + y) par seconde application de la première règle
- *(*(tab + x) + y) ⇔ *(*(x + tab) + y) par commutativité entre "tab" et "x"
- *(*(x + tab) + y) ⇔ *(y + *(x + tab)) par commutativité entre "*(x + tab) et "y"
- *(y + *(x + tab)) ⇔ *(y + x[tab]) par application inverse de la première règle sur "*(x + tab)"
- *(y + x[tab]) ⇔ y[x[tab]] par seconde application inverse de la première règle

On peut ainsi écrire "tab[x][y]…[z]" ou bien "z[…[y[x[tab]]]…]" dans le cas de tableaux à plusieurs dimensions

6) Danger du pointeur

Le vrai et seul danger d'un pointeur est d'aller lire ou modifier la zone pointée… sans la connaître. En effet, définir un pointeur (c'est à la base une variable) sans l'initialiser revient à le remplir d'une valeur imprévisible. Ensuite, aller voir ou modifier à l'adresse correspondant à cette valeur **n'est pas du tout une bonne idée** !!!
Exemple :

```
float *pt;                    // Que contient "pt" ???
*pt=3.1416;                   // Où va aller la valeur "3.1416" ???
```

Voici un schéma de la mémoire :

Adresse	Nom	Contenu
0x1000	pt	xxx
xxx	*pt	3.1416

A–t-on le droit d'écrire à l'adresse "*xxx*" ? Cette adresse n'est-elle pas par hasard l'adresse d'une autre variable ?
En général, ce genre de manipulation conduit à 2 cas :

- le programme plante immédiatement. On sait au moins qu'il y a un problème et on peut le chercher. Avec un peu d'habitude, on le trouve généralement assez rapidement.
- le programme fonctionne sans problème apparent car il se trouve que l'adresse "*xxx*" est libre et accessible. C'est le cas le plus grave car le problème existe mais ne se montre pas pendant 1, 10, 100 essais. Puis, un jour, le programmeur rajoute une variable innocente mais qui utilise l'adresse "xxx" et le programme plante immédiatement. Mais l'erreur est depuis longtemps enfouie dans la masse des lignes de codes.

Pour éviter ce genre d'erreur, il ne faut jamais utiliser "*pt" si on n'a pas auparavant rempli "pt" avec une adresse cohérente (connue)

7) Tableaux de pointeurs

Un pointeur n'étant qu'un nombre, rien ne l'empêche d'être stocké avec d'autres nombres de sa catégorie, dans un tableau. On nomme un tel tableau un "tableau de pointeurs".
L'utilité d'un tel tableau est courante dès qu'on parle de chaînes. Une chaîne de caractère étant un tableau de caractères, c'est déjà un pointeur. Et si on veut manipuler et stocker plusieurs chaînes dans un tableau, il devient obligatoire d'utiliser un "tableau de pointeurs". Cela permet, en plus, de gagner de la place surtout quand on travaille sur des chaînes de longueurs inégales.
Exemple de tableau à deux dimensions :

```
// Tableau des jours de la semaine
char semaine[7][9]={                                    // Tableau de 7 lignes sur 9 colonnes
        {'l', 'u', 'n', 'd', 'i', 0},                   // 6 colonnes utilisées sur 9
        {'m', 'a', 'r', 'd', 'i', 0},                   // 6 colonnes utilisées sur 9
        {'m', 'e', 'r', 'c', 'r', 'e', 'd', 'i', 0},    // 9 colonnes utilisées sur 9
        {'j', 'e', 'u', 'd', 'i', 0},                   // 6 colonnes utilisées sur 9
        {'v', 'e', n', 'd', 'r', 'e', 'd', 'i', 0},     // 9 colonnes utilisées sur 9
        {'s', 'a', m', 'e', 'd', 'i', 0},               // 7 colonne sutilisées sur 9
        {'d', 'i', 'm', 'a', 'n', 'c', 'h', 'e', 0}     // 9 colonnes utilisées sur 9
};
```

Représentation de la mémoire

Adresse	Nom	Contenu
0x10	semaine	0x1000
0x1000	semaine[0][0]	'l'
0x1001	semaine[0][1]	'u'
0x1002	semaine[0][2]	'n'
0x1003	semaine[0][3]	'd'
0x1004	semaine[0][4]	'i'
0x1005	semaine[0][5]	0
0x1006	semaine[0][6]	*non utilisé mais réservé (perdu)*
0x1007	semaine[0][7]	*non utilisé mais réservé (perdu)*
0x1008	semaine[0][8]	*non utilisé mais réservé (perdu)*
0x1009	semaine[1][0]	'm'
0x1010	semaine[1][1]	'a'
0x1011	semaine[1][2]	'r'
0x1012	semaine[1][3]	'd'
0x1013	semaine[1][4]	'i'
0x1014	semaine[1][5]	0
0x1015	semaine[1][6]	*non utilisé mais réservé (perdu)*
0x1016	semaine[1][7]	*non utilisé mais réservé (perdu)*
0x1017	semaine[1][8]	*non utilisé mais réservé (perdu)*
0x1018	semaine[2][0]	'm'
etc.	etc.	etc.

Dans cet exemple, on perd 11 octets sur 64 ; c'est à dire 17% de la mémoire réservée. Cela peut paraître insignifiant au regard des puissances actuelles de nos machines mais transposée à l'échelle d'un vaste projet, 17% de mémoire perdue ou gagnée peuvent faire la différence entre un programme médiocre et un programme de qualité. Surtout si la zone est elle-même dupliquée plusieurs fois (dans un tableau, dans une fonction récursive, etc.)

Exemple de tableau de pointeurs :

```
// Tableau des jours de la semaine
char *semaine[7]={                      // Tableau de 7 pointeurs sur un caractère
        "lundi",                        // Pointeur
        "mardi",                        // Pointeur
        "mercredi",                     // Pointeur
        "jeudi",                        // Pointeur
        "vendredi",                     // Pointeur
        "samedi",                       // Pointeur
        "dimanche"                      // Pointeur
};
```

Représentation de la mémoire

Adresse	Nom	Contenu
0x10	semaine	0x1000
0x1000	semaine[0]	0x2000
0x1001	semaine[1]	0x3000
0x1002	semaine[2]	0x4000
0x1003	semaine[3]	0x5000
0x1004	semaine[4]	0x6000
0x1005	semaine[5]	0x7000
0x1006	semaine[6]	0x8000
0x1007	…	*utilisé par d'autres variables*
…	…	*utilisé par d'autres variables*
0x2000	semaine[0][0]	'l'
0x2001	semaine[0][1]	'u'
0x2002	*(semaine[0] + 2)	'n'
0x2003	semaine[0][3]	'd'
0x2004	semaine[0][4]	'i'
0x2005	semaine[0][5]	0
…	…	*utilisé par d'autres variables*
0x3000	semaine[1][0]	'm'
0x3001	semaine[1][1]	'a'
0x3002	*(*(semaine + 1) + 2)	'r'
etc.	etc.	etc.

Dans cet exemple, on ne perd aucune partie de la mémoire. Et la manipulation de telles structures ne pose pas de soucis quand on y est habitué.

Remarque : Même dans le cas de tableau de pointeurs, on peut encore remplacer l'opérateur "*****" par des crochets. De plus, cette technique de tableau contenant l'adresse d'autres tableaux peut être généralisée pour représenter des tableaux à plusieurs dimensions. Malheureusement, de tels ensembles deviennent vite difficile à schématiser donc il convient de programmer le langage C avec rigueur afin de ne pas générer d'incohérences.

8) Fonctions, paramètres et pointeurs

a) Modification d'une variable passée à une fonction

Il a été vu, lors de l'apprentissage des fonctions en C, que celles-ci ne pouvaient pas modifier la variable qu'elle reçoit en paramètre car la fonction n'en reçoit qu'une copie.
Cependant, il est possible de passer à une fonction **l'adresse d'une variable**. La fonction copiera cette adresse dans une zone temporaire mais **saura à quel emplacement mémoire se trouve la variable d'origine et pourra donc la modifier**.

Exemple de passage d'adresse à une fonction :

```
int main(void)
(
        float var=2.71828;                  // Variable à modifier

        // Appel de la fonction
        modif(&var);                        // La fonction "modif" reçoit l'adresse de "var"
        printf("var=%f\n", var);
}
```

Représentation de la mémoire

Adresse	Nom	Contenu
0x1000	var	2.71828

La fonction va recevoir la valeur "0x1000". Cette valeur étant un nombre entier, on peut se dire qu'elle reçoit un "int" qu'on appellera par exemple "pt". Cependant, comme pour le premier pointeur du chapitre, la fonction va devoir aller toucher **ce qui est pointé par "pt"** donc va devoir aller toucher "***pt**". Or, le langage ne saura pas ce qu'il y a à cet endroit si on ne le lui a pas dit. Et à l'endroit représenté par "***pt**", il y a un "**float**".
Exemple de la fonction "modif" :

```
void modif(
        float *pt)                                  // Pointeur sur un nombre "float"
(
        // Modification de la valeur
        *pt=3.1416;                                 // Modification de ce qui est "pointé par pt"
}
```

Représentation de la mémoire

Adresse	Nom	Contenu
0x1000	var	2.71828
0x5000	pt	0x1000

Puisque la fonction reçoit la valeur "0x1000" et qu'elle connaît le type de ce qu'il y a à cet endroit, elle peut le modifier. C'est pour cela que la fonction "***scanf()***" doit impérativement recevoir l'adresse de la variable à saisir. Et c'est aussi pour cela qu'on ne passe pas l'adresse d'une chaîne à la fonction "***scanf***" ; parce qu'une chaîne est déjà un tableau et que nommer une chaîne consiste à nommer l'adresse de son premier élément.

Remarque :

Si "var" est de type "float", alors "&var" est de type "float étoile"

On peut arriver aussi à cette remarque en utilisant l'astuce suivante :
"var" est de type "float" donc "*&var" est de type "float" donc "&var" est de type "float *" (si on ôte l'étoile d'un coté de l'équation, on la replace de l'autre coté).

b) Doit-on mettre "&" ou "*" ?

Arrivé à ce niveau, le débutant en C se heurte assez rapidement à un dilemne. La fonction doit recevoir un "**float étoile**" et le programme possède déjà une variable "pt" déclarée comme "float étoile" : "**float *pt**".
Le but étant que la fonction reçoive la valeur de "**pt**". Souvent, le débutant se dit "*elle reçoit un float étoile, je lui passe alors "étoile pt*" comme dans l'exemple suivant :

```
void modif(
        float *param)                   // Paramètre reçu par la fonction
(
        // Modification de la valeur
        *param=2.71828;                 // Modification de ce qui est "pointé par param"
}

int main(void)
{
        // Déclaration des variables
        float pi=3.1416;                // Variable "pi"
        float *pt=&pi;                  // Variable "pt" contenant l'adresse de "pi"

        // Programmation
        modif(*pt);                     // Mauvais
        modif(&pt);                     // Très mauvais
        modif(pt);                      // Correct
        modif(&pi);                     // Correct
}
```

Examinons à nouveau la mémoire

Adresse	Nom	Contenu
0x1000	pi	3.1416
0x2000	pt	0x1000
0x5000	param	?

On ne connaît pas le contenu de "param". Mais notre désir est qu'il reçoive la valeur "0x1000" afin que la fonction "modif" puisse aller modifier la valeur "3.1416".

- si on passe à la fonction "*pt", on lui passe "3.1416" ; ce qui n'est pas le but. D'ailleurs, "*pt" est de type "float" alors que la fonction attend un "float étoile"
- si on lui passe "&pt", on lui passe "0x2000" ; ce qui n'est toujours pas bon. De plus, "pt" étant de type "float étoile", "&pt" est de type "float étoile étoile" et ne correspond pas non plus au type attendu "float étoile".
- si on lui passe "pt", on lui passe "0x1000" ; ce qui est correct. Et "pt" est de type "float étoile", il correspond au type attendu par la fonction.
- si on lui passe "&var", on lui passe aussi "0x1000" ce qui est toujours correct. Et "&var" est aussi du type attendu par la fonction "float étoile".

c) Fonction renvoyant un pointeur

Un pointeur étant une valeur comme une autre, la fonction peut renvoyer cette valeur. Il suffit alors de déclarer la fonction du type de la valeur qu'elle renvoie. Et ainsi, on peut avoir des fonctions de type "char étoile", "int étoile", ou "double étoile" selon le cas.
Exemples de déclarations diverses :

```
char *f1(void);                 // "f1" renvoie un pointeur sur un caractère"
int *f2(void);                  // "f2" renvoie un pointeur sur un entier
double *f3(void);               // "f3" renvoie un pointeur sur un réel double précision
```

9) Pointeur sur rien – Pointeur sur tout

Ces types de pointeurs vus jusqu'à présent viennent s'enrichir d'un type nouveau : le pointeur universel "void étoile". Il s'agit d'un pointeur pouvant pointer sur n'importe quoi non prévu à l'avance. Puisque le type "void" était libre et qu'un pointeur sur "vide" n'avait aucune signification, le type "void étoile" est devenu synonyme de "pointeur sur tout".

Evidemment, pour utiliser un "pointeur sur tout", il faut le transformer ("caster") en "*pointeur sur le type désiré*" avant d'y appliquer l'opérateur "*pointé par*"
Exemple :

```
main()
{
	// Déclaration des variables
	float pi=3.1416;				// Variable "pi"
	void *pt;					// Pointeur universel "pt"

	// Puisque "pt" est universel, je peux y mettre n'importe quoi
	pt=&pi;					// J'y mets donc l'adresse de "pi"

	// Modification de ce qui est pointé par "pt" mais obligation de le "caster"
	*(float*)pt=2.71828;			// "pt" est "casté" en "pointeur sur float"
}
```

10) Exercices

Ecrire une fonction qui compte et renvoie le nombres de caractères de la chaîne qu'elle reçoit en argument (une chaine est un tableau de caractères avec un caractère supplémentaire valant zéro pour indiquer où se termine la chaine).

```
// Prototype fonction "ChaineLen" recevant une chaîne à compter
unsigned long ChaineLen(char *);

// Fonction principale du programme
int main(void)
{
	// Déclaration des variables
	char chaine[100];				// Chaîne saisie au clavier

	// Saisie de la chaîne à compter
	printf("Entrez la chaîne : ");
	scanf("%s", chaine);

	// Affichage
	printf("La taille de la chaîne %s est %lu\n", chaine, ChaineLen(chaine));

	// Fin du programme
	return 0;
}

// Fonction "ChaineLen"
unsigned long ChaineLen(
	char *ch)				// Pointeur sur la chaîne reçue
{
	// Déclaration des variables
	unsigned long cpt;			// Compteur de chaîne

	// La fonction va incrémenter le pointeur reçu jusqu'à ce que le caractère pointé soit égal à 0
	// Heureusement que les octets d'un tableau sont contigüs en mémoire
	// Heureusement que l'un de ces octets vaut "0" pour permettre de trouver la fin
	cpt=0;

	// Tant que le pointé n'est pas égal à "0"
	while (*ch != '\0')			// Il est plus courant d'utiliser '\0' au lieu de 0
	{
		// Incrément du compteur
		cpt++;

		// Décalage du pointeur
		ch++;
	}
	// On aurait pu écrire toute cette boucle en une ligne "for (cpt=0; *ch != '\0'; cpt++, ch++);"

	// On a perdu "ch", le début de chaîne, mais ce n'est pas grave, la fonction est finie
	// Ici, "cpt" contient la longueur de la chaîne
	return cpt;
}
```

Ecrire une fonction qui compare deux chaînes. Elle doit renvoyer :
✓ un nombre < 0 si chaîne1 < chaîne2
✓ un nombre > 0 si chaîne1 > chaîne2
✓ 0 si les deux chaînes sont égales (tous ses octets égaux un à un)
La valeur absolue du nombre renvoyé doit correspondre à la position du caractère qui diffère

```
// Prototype fonction "ChaineCompar" recevant deux chaînes à comparer
long ChaineCompar(char *, char *);

// Fonction principale du programme
int main(void)
{
	// Déclaration des variables
	char chaine1[100];			// Chaîne1 saisie au clavier
	char chaine2[100];			// Chaîne2 saisie au clavier
	long res;				// Résultat comparaison

	// Saisie de la chaîne1
	printf("Entrez la chaîne1 : ");
	scanf("%s", chaine1);

	// Saisie de la chaîne2
	printf("Entrez la chaîne2 : ");
	scanf("%s", chaine2);

	// Comparaison
	res=ChaineCompar(chaine1, chaine2) ;

	// Affichage
	if (res == 0)
		printf("%s identique à %s\n", chaine1, chaine2);
	else
	{
		if (res > 0)
			printf("%s plus grand que %s à partir de %lu\n", chaine1, chaine2, res);
		else
			printf("%s plus petit que %s à partir de %lu\n", chaine1, chaine2, -res);
	}

	// Fin du programme
	return 0;
}

// Fonction "ChaineCompar"
long ChaineCompar(
	char *ch1,			// Pointeur sur la chaîne 1 à comparer
	char *ch2)			// Pointeur sur la chaîne 2 à comparer
{
	// Déclaration des variables
	unsigned long i;		// Indice de boucle

	// On considère qu'elles sont égales par défaut et on cherchera une différence
	i=0;
	do {
		i++;			// "i" prend la position du caractère testé

		if (*ch1 < *ch2)	// Si caractère ch1 < caractère ch2
			return -i;	// Sortie de fonction avec position négative
```

```
                if (*ch1 > *ch2)                    // Si caractère ch1 > caractère ch2
                        return i;                   // Sortie de fonction avec position positive

        while (*ch1++ != '\0' && *ch2++ != '\0');
        // Tant que fin chaîne non atteinte ("*ch++" renvoie valeur "*ch" puis incrémente "ch")

        // Les deux chaînes sont égales – Sortie de fonction
        return 0;
}
```

Reécrire la fonction "main" de l'exercice précédent mais en utilisant un tableau de deux chaînes

```
// Fonction principale du programme
int main(void)
{
        // Déclaration des variables
        char chaine[2][100];                        // Tableau de deux chaînes saisies au clavier
        unsigned short i;                           // Indice de boucle
        long res;                                   // Résultat comparaison

        // Boucle de saisie
        for (i=0; i < 2; i++)
        {
                // Saisie de la chaîne
                printf("Entrez la chaîne %hu : ", i + 1);
                scanf("%s", chaine[i]);
        }

        // Comparaison
        res=ChaîneCompar(chaine[0], chaine[1]) ;

        // Affichage
        if (res == 0)
                printf("%s identique à %s\n", chaine[0], chaine[1]);
        else
        {
                if (res > 0)
                        printf("%s plus grand que %s à partir de %ul\n", chaine[0], chaine[1], res);
                else
                        printf("%s plus petit que %s à partir de %ul\n", chaine[0], chaine[1], -res);
        }

        // Fin du programme
        return 0;
}
```

Ecrire une fonction qui renvoie le nom de base d'un chemin Unix.
Le nom de base est tout ce qui suit le dernier "/" (slash)
Cas particulier : Le nom de base de "/" est "/"

```
// Prototype fonction "BaseName" recevant un nom Unix
char *BaseName(char *);

// Fonction principale du programme
int main(void)
{
	// Déclaration des variables
	char nom[100];				// Nom saisi au clavier

	// Saisie du nom
	printf("Entrez le nom : ");
	scanf("%s", nom);

	// Affichage
	printf("Le nom de base de %s est %s\n", nom, BaseName(nom));

	// Fin du programme
	return 0;
}

// Fonction "BaseName"
char *BaseName(
	char *ch)				// Pointeur sur le nom à traiter
{
	// Déclaration des variables
	char *pt;				// Second pointeur sur chaîne

	// Cas particulier : Le nom de base de "/" est "/"
	if (ch[0] == '/' && ch[1] == '\0')
		return ch;

	// On ne sait pas combien de "/" dans la chaine donc on commence par chercher le '\0' final
	for (pt=ch; *pt !='\0'; pt++);
	// ici, "pt" est positionné sur le '\0' final (dernier octet de la chaîne)

	// Recherche d'un "/" en faisant reculer "pt" au maximum jusqu'au début de chaîne
	while (pt >= ch && *pt != '/')		// Il faut tester "pt" avant de tester "*pt" !!!
		pt--;

	// Ici, soit "pt" est sur le dernier "/"; soit "pt" est juste avant le début de la chaîne
	// Dans tous les cas, il faut renvoyer l'adresse qui suit "pt"
	return (pt + 1);

	//
		La variable renvoyée "pt + 1" étant de type "char étoile",
			la fonction est de type "char étoile"

		De plus, même si "pt" arrive avant le début de "ch" et passe donc
			dans une zone interdite; comme on teste "pt" avant de tester "étoile pt"
			la boucle s'arrêtera avant que le programme ne plante !!!
	                                                                        */
}
```

11) Pointeur sur fonction

a) Généralités

Une fonction a aussi une adresse. Il s'agit de l'adresse mémoire du début de son code d'instructions. Puisque cette fonction a une adresse, rien n'empêche de récupérer cette adresse dans une variable. Il suffit juste de savoir comment s'appelle cette adresse et comment définir la variable.
L'adresse de la fonction est stockée dans une variable qui porte le nom de la fonction.

```
unsigned long carre(short nb)
{
        return (nb * nb);
}
```

Examinons la mémoire

Adresse	Nom	Contenu
???	carre	0x1000
0x1000	carre()	Code de la fonction "carre"

Pour savoir comment définir la variable "pt" devant recevoir l'adresse "0x1000", il faut raisonner comme on l'a fait jusqu'à présent :

- "pt" est un pointeur donc on aura forcément au moins une "étoile" dans son nom De plus, le langage a besoin de savoir ce qu'il y a à "*pt". Donc, il faut définir "*pt".
- *pt est une fonction. Cette fonction renvoie un "unsigned long" et reçoit un "short"

Il en résulte la syntaxe suivante :

```
unsigned long (*pt)(short);
```

Attention : L'emploi des parenthèses autour de "*étoile pt*" est **obligatoire** à cause des priorités des opérateurs. Si vous écriviez "unsigned long *pt(short)" ; vous déclareriez une fonction nommée "pt" recevant un "short" et renvoyant un "unsigned long étoile" (pointeur sur un long non signé). Alors que votre but est de déclarer un "pointeur sur une fonction dont on ne connaît pas le nom" recevant un "short" et renvoyant un "unsigned long".

b) A quoi ça sert

Il s'agit en effet d'utilisation avancées du langage dont on peut se servir. On l'utilise dans le cadre de fonctions "génériques" ou "universelles" ; c'est à dire des fonctions qui font un traitement quelconque (tri par exemple) sur des éléments non connus lors de la programmation du traitement.

Il faut bien se rappeler qu'on est dans un environnement de programmation coopératif. Certains programmeurs créent des algorithmes que d'autres utiliseront. Comme les premiers ne savent pas sur quels objets s'appliqueront leurs algorithmes, ils se contentent de programmer le traitement général qui travaillera sur des objets "génériques" grace à des pointeurs universels (void *).
Et les seconds ont ainsi à leur disposition un algorithme utilisable pour leurs objets personnels. Ils n'ont plus alors qu'à créer la fonction de traitement de leur objet (puisqu'ils connaissent leurs objets) et passer un pointeur sur cette fonction à l'algorithme général.
Ce dernier, lorsqu'il sera exécuté, pourra ainsi traiter chaque objet qu'il aura à manipuler puisqu'il aura à sa disposition l'adresse de la fonction manipulatrice.

c) Exemple

Un exemple des plus connus des programmeurs du C est la fonction "qsort" qui utilise l'algorithme de tri rapide par pivots (quick sort).
Cette fonction peut trier un tableau contenant n'importe quel type d'élément. Le programmeur qui désire l'utiliser doit juste passer à la fonction

- le tableau ou l'adresse du premier élément (la fonction sait ainsi quoi trier)
- le nombre d'éléments du tableau (la fonction sait ainsi combien il faut trier)
- la taille d'un élément en octets (la fonction peut donc permuter deux éléments si besoin octet par octet)
- un pointeur sur une fonction que doit écrire l'utilisateur de "qsort". Ce dernier connaît les objets qu'il veut trier donc il sait les comparer. Cette fonction a donc pour mission de comparer deux éléments et doit impérativement
  - recevoir un pointeur sur chacun des deux éléments à comparer
  - renvoyer un "int" de valeur –1, 0 ou 1 selon que l'élément 1 est plus petit, égal ou plus grand que l'élément 2

L'exemple suivant va trier un tableau de 10 flottants

```
// Prototype fonction "compar" recevant deux pointeurs sur deux flottants
static int Compar (float*, float*);

// Fonction principale du programme
int main(void)
{
	// Déclaration des variables
	float tab[10];				// Tableau de 10 flottants à trier
	float *pt;				// Pointeur sur le tableau
	unsigned short i;			// Indice de boucle

	// Saisie des flottants
	for (i=0, pt=tab; i < 10; i++, pt++)
	{
		printf("Entrez le nombre %hu : ", i + 1);
		scanf("%f", pt);		// Pas "&pt" puisque "pt" ⇔ "&tab[i]"
	}

	// Tri du tableau – Un seul appel suffit
	qsort(tab, 10, sizeof(float), Compar);
	// La fonction utilisera le pointeur sur "Compar()" pour comparer les éléments du tableau

	// Affichage du tableau trié
	for (i=0, pt=tab; i < 10; i++, pt++)
		printf("Le nombre %hu est %f\n ", i, *pt);

	// Fin du programme
	return 0;
}

// Fonction "Compar" à écrire par le programmeur
static int Compar(
	float *elem1,			// Elément 1 à comparer
	float *elem2)			// Elément 2 à comparer
{
	// Comparaison et renvoi
	if (*elem1 < *elem2)
		return -1;
	if (*elem1 > *elem2)
		return 1;
	return 0;
	// Il suffit d'inverser chaque test d'inégalité pour inverser le tri du tableau !!!
}
```

d) Exercice

Se mettre dans la peau d'un programmeur n°1 et écrire une fonction "générique" qui balaye un tableau d'entiers. Elle n'a pas besoin de savoir quoi faire de l'entier balayé car elle recevra en dernier paramètre un pointeur sur la fonction qui traite cet entier.

```
// Fonction générique de balayage
unsigned short Balaye(
        int *tab,                                   // Pointeur sur le tableau à balayer
        unsigned short nb,                          // Nombre d'éléments du tableau
        void (*pt_trt)(unsigned short, int))        // Pointeur sur la fonction de traitement
{
        // Déclaration des variables
        unsigned short i;                           // Indice de boucle

        // Boucle sur le tableau
        for (i=0; i < nb; i++)
        {
                // Traitement de l'élément – A programmer par celui qui utilisera cette fonction
                (*pt_trt)(i, tab[i]);
        }

        // Renvoi d'une valeur qui peut être utile (facultatif)
        return nb;
}
```

Se mettre dans la peau d'un programmeur n°2 et utiliser la fonction écrite précédemment

```
// Prototype fonction "Traitement" recevant un entier court non signé et un entier
static void Traitement (unsigned short, int);

// Fonction principale
int main(void)
{
        // Déclaration des variables
        unsigned short val;                         // Valeur quelconque
        static int tab[]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10};   // Tableau à traiter

        // Balayage du tableau
        val=Balaye(tab, 10, Traitement);
        printf("Ma fonction a balayé %hu éléments de tab\n", val);
        return 0;
}

// Fonction de traitement
static void Traitement(
        unsigned short ind,                         // Indice élément à traiter
        int elem);                                  // Elément à traiter
{
        // Affichage de l'élément
        printf("Elément %hu = %d\n", ind + 1, elem);
}
```

XI) LES TYPES DERIVES

Il s'agit des types composés de types simples ou d'autres types dérivés.

1) Les structures

a) Généralités

Une structure est une collection de données regroupées dans une même entité logique. Elles permettent de créer des types complexes à partir de différents types déjà connus et de les regrouper sous un même nom.
Exemple : Une date est composée d'un jour, d'un mois et d'une année. Il est donc plus judicieux, si on veut gérer plusieurs dates de créer une structure "s_date" et de gérer plusieurs variables de cette structure unique.

Il est impossible de manipuler (affecter, comparer) une structure dans son intégralité. Les seules opérations permises sur les structures sont donc :

- l'accès à ses membres (les variables qui la composent si ces variables sont de type simple)
- la récupération de son adresse

Par contre, les membres d'une structure sont utilisables comme n'importe quelle variable et sont donc manipulables comme tels ; à condition que ces membres eux-mêmes ne soient ni "tableau" ni "structure".

Lorsqu'une structure est créée, tous les membres de la structure ont des zones contiguës en mémoire.

b) Définition d'une structure

La déclaration d'une structure se fait sous la forme suivante :

```
struct [<nom struct> ] {
        type1 <nom élément1>, <nom élément2>;
        type2 <nom élément3>;
        etc…
} [var1] [,var2 …];
```

- Le nom de la structure n'est pas obligatoire. Il permet simplement de nommer la structure afin de pouvoir réutiliser son format. Dans ce cas, la chaîne "struct <nom struct>" devient un type à part entier qu'on peut utiliser avec un nom de variable pour créer une variable de ce type.
- Les variables associées ne sont pas non-plus obligatoires. On peut en effet créer une structure avec un nom d'une part et définir ensuite des variables de cette structure d'autre part.

Exemple :

```
struct s_date {                         // Création d'une structure pour gérer la date"
        unsigned char jj;               // Jour de la date
        unsigned char mm;               // Mois de la date
        unsigned short aa;              // Année de la date
} naissance;                            // Variable "naissance" de type "struct s_date"

struct s_date jour;                     // Variable "jour" de type "struct s_date"
```

L'initialisation d'une structure se fait comme pour un tableau. D'ailleurs, comme pour ce dernier, l'initialisation globale ne peut se faire que lors de la définition de la variable :
De plus, il est assez courant d'associer un nom de structure avec l'instruction "*typedef*". Cela évite d'avoir à écrire le mot "*struct*" devant chaque variable que l'on veut déclarer.

Exemple :

```
typedef struct {                        // Création d'une structure pour gérer la date"
        unsigned char jj;               // Jour de la date
        unsigned char mm;               // Mois de la date
        unsigned short aa;              // Année de la date
        char *jour;                     // Chaîne correspondant au jour
} t_date;                               // Le type "t_date" correspond à cette structure

// Définition et initialisation d'une variable
t_date today= {                         // N'oubliez pas l'accolade ouvrante "{"
        04,                             // Cette valeur ira dans le champ "jj"
        02,                             // Cette valeur ira dans le champ "mm"
        2013,                           // Cette valeur ira dans le champ "aa"
        "lundi"                 // L'adresse de cette chaîne ira dans le champ "jour"
};                                      // N'oubliez pas l'accolade fermante "}"
```

Il est bien entendu possible d'inclure des structures dans d'autres structures. L'écriture de tels ensembles peut se faire en une ou plusieurs étapes. Dans ce dernier cas, il faut alors définir la structure la plus "intérieure" avant de définir la structure la plus "extérieure".
Exemple :

```
typedef struct {
        long x, y;                      // Abscisse et ordonnée du point
} t_point;                              // Type permettant de gérer un point

typedef struct {
        t_point bg,                     // Coin bas-gauche du carré
        long cote,                      // Cote du carré
} t_carre;                              // Type permettant de gérer un carré

t_carre trigo={                         // Création variable "trigo" de type "t_carre"
        {0, 0},                         // Coordonnées du coin bas-gauche
        10,                             // Ceci est son coté
};                                      // Variable "trigo" créée et initialisée
```

Il est plus judicieux de déclarer les structures de travail en dehors de tout bloc afin qu'elles soient connues de tout le programme. Par contre, le choix de la visibilité et de la durée de vie des variables structurées reste à la discrétion du programmeur

c) L'accès aux membres d'une structure

L'accès aux membres d'une structure se fait avec l'opérateur "**.**" (un "point") si on utilise une variable structurée ; et par l'opérateur "**->**" (un "moins" suivi d'un "supérieur") si on utilise un pointeur sur cette structure ; et ceci quel que soit le membre visé (variable simple ou pointeur).
Bien entendu, on peut aussi demander l'adresse d'un membre d'une variable structurée et l'adresse d'un membre d'un pointeur sur cette variable.
Pour bien voir toutes les possibilités, on va utiliser une structure permettant de gérer un livre (titre et taille).
Ce type comporte une variable simple (la taille) et un tableau qui est aussi un pointeur (le titre) :

```
typedef struct {
        unsigned long size;                 // Taille du livre (variable simple)
        char titre[20],                     // Titre du livre (pointeur)
} t_livre;                                  // type permettant de gérer un livre

t_livre bouquin;                            // Variable "bouquin" de type "t_livre"
t_livre *pt=&bouquin;                       // Pointeur "pt" sur la variable "bouquin"
```

En utilisant la variable "bouquin" :

Elément choisi	Membre	Accès au membre	Accès à l'adresse du membre
Taille du livre	size	Emploi de l'opérateur "**.**" tout simplement ☞ bouquin**.**size	Il suffit de rajouter un "**&**" devant l'accès au membre : ☞ &bouquin**.**size
"x + 1ième" lettre du titre	titre[x]	Emploi à nouveau de l'opérateur "**.**" tout simplement ☞ bouquin**.**titre[x] Mais n'oublions pas les règles d'équivalence "tab[x] ⇔ *(tab + x)" ce qui donne aussi : ☞ *(bouquin**.**titre + x)	Comme pour une variable simple : ☞ &bouquin**.**titre[x] Mais n'oublions pas les règles d'équivalence "&tab[x] ⇔ tab + x" ce qui donne aussi : ☞ bouquin**.**titre + x
La chaîne correspondant au titre dans sa globalité	titre	Ce membre est utilisé alors comme tout autre membre ☞ bouquin**.**titre	Comme pour une variable simple (bien que cela soit peu courant, car cela revient à demander l'adresse d'une adresse) ☞ &bouquin**.**titre

En utilisant la variable "pt" :

Elément choisi	Membre	Accès au membre	Accès à l'adresse du membre
Taille du livre	size	Il s'agit du membre "size" du pointé "*pt" ; ce qui donne : ☞ (*pt)**.**size Les parenthèses "(*pt)" sont obligatoires à cause de la priorité de l'opérateur "**.**". Cette écriture assez lourde est avantageusement remplacée par l'opérateur "**->**" : ☞ pt**->**size	Il suffit de rajouter un "**&**" devant l'accès au membre : ☞ &pt**->**size
"x + 1ième" lettre du titre	titre[x]	Comme pour "size" sans oublier les règles d'équivalence : ☞ pt**->**titre[x] ☞ *(pt**->**titre + x)	Comme pour une variable simple sans oublier les règles d'équivalence : ☞ &pt**->**titre[x] ☞ pt**->**titre + x
La chaîne correspondant au titre dans sa globalité	titre	Ce membre est utilisé alors comme tout autre membre ☞ pt**->**titre	Comme pour une variable simple (bien que cela soit peu courant, car cela revient à demander l'adresse d'une adresse) ☞ &pt**->**titre

2) Les unions

a) Généralités

L'union permet de déclarer des objets polymorphes : objets qui peuvent changer de forme mais qui n'ont à un moment donné qu'une forme et une seule.
L'union permet de faire partager un même emplacement mémoire par des variables de types différents. Elle permet :

- d'économiser des emplacements mémoire
- d'interpréter de façons différentes un même motif binaire

Comme pour une structure, il est impossible de manipuler (affecter, comparer) une union dans son intégralité. Les seules opérations permises sur les unions sont donc :

- l'accès à ses membres
- la récupération de son adresse.

Par contre, les membres d'une union sont utilisés comme n'importe quelle autre variable et sont donc manipulables comme tels ; à condition que ces membres ne soient ni "tableau" ni "structure" ni "union". Mais évidemment, comme tous les membres sont sur le même emplacement mémoire, on ne peut en utiliser qu'un seul à la fois (toute affectation d'un membre écrase tous les membres de l'union).

Lors de la déclaration de la variable, l'union de données ne réserve en mémoire que la place nécessaire au stockage de la plus grande des entités qu'elle contient.
Mis à part ces détails, on utilise une union exactement comme une structure.

b) Définition d'une union

La déclaration d'une union se fait sous la forme suivante :

```
union [<nom union> ] {
        type1 <nom élément1>, <nom élément2>;
        type2 <nom élément3>;
        etc…
} [var1] [,var2 …];
```

c) L'accès aux membres

L'accès aux membres se fait comme pour les structures avec l'opérateur "**.**" ou "**->**" selon qu'il s'agisse d'une variable ou d'un pointeur sur une union.

Exemple :

```
union {
        long nb;                        // Nombre format long
        char elem[4];                   // Chaque octet du nombre
} analyse;                              // Variable "analyse" permettant d'analyser un "long"

// Remplissage du nombre
printf("Entrez un nombre :");
scanf("%ld", &analyse.nb);

// Affichage de ses octets
for (i=0; i < 4; i++)
        printf("L'octet %hu vaut %d\n", i + 1, analyse.elem[i]);
```

3) Les énumérations

Une énumération est une liste de mots choisis par le programmeur qui seront alors associés à des valeurs entières contantes.

```
enum [<nom enum>] {mot1 [=valeur][, mot2 [=valeur]]…} [var1] [,var2 …];
```

La liste des mots, séparée par des virgules, est choisie par le programmeur. Si ce dernier n'y associe pas de valeur, celles-ci seront automatiquement crées par le compilateur à partir de "0" et croissantes de "1" en "1".
Il est bien sûr possible de n'associer des valeurs particulières qu'à quelques mots et laisser le compilateur remplir les mots restés libres. Cependant, les valeurs associées aux mots ne sont pas modifiables dans le code des programmes. Enfin, deux mots distincts peuvent être associés à une même valeur.
Les variables de type énuméré restent cependant des variables entières et sont manipulables comme tel. Jusqu'à une certaine époque, on ne pouvait utiliser les mots énumérés qu'avec des variables du type énuméré correspondant. Aujourd'hui, la norme accepte l'utilisation d'un mot énuméré dans toute variable de type entier.

Exemple :

```
// Exemples d'énumérations
enum e_semaine {lundi=1, mardi, mercredi, jeudi, vendredi, samedi=11, dimanche};
enum e_niveau {bas=500, moyen=1000, haut=1500, min=500, max=1500};

// Fonction principale
int main(void)
{
	// Déclaration des variables
	enum e_semaine jour;

	// Saisie et évaluation du jour
	printf("Entrez un nombre : ");
	scanf("%d", &jour);
	if (jour == samedi || jour == dimanche)
		printf("%d correspond au week-end\n", jour);
	else
		printf("%d correspond à la semaine\n", jour);

	// Fin du programme
	return 0;
}
```

L'emploi de tels objets reste assez faible car il est possible d'obtenir le même résultat en utilisant le pré processeur.

XII) LE PRE PROCESSEUR

Le pré processeur est la première phase de la compilation. Son rôle est de transformer un source ".c" du langage C en source ".i" directement interprétable par le compilateur. Mais le programmeur a à sa disposition des directives pré processeur permettant de configurer et régler la traduction du ".c" en ".i"

1) La directive "#define"

a) Utilisation

La directive "***#define***" permet de créer des pseudo constantes et des pseudo fonctions. Ce sont des chaînes associées à des valeurs ou des instructions et directement remplacées par celles-ci lors de la transformation du source ".c" en ".i".
Ces chaînes sont appelées "**macro définition**" et sont, par convention, écrites en majuscules.

Exemple ".c" :

```
#define TAILLE          100
#define CARRE(n)        n * n

int main(void)
{
        int tab[TAILLE];
        int i;

        for (i=0; i < TAILLE; i++)
                tab[i]=CARRE(i);
}
```

Code généré dans le ".i" :

```
int main(void)
{
        int tab[100];
        int i;

        for (i=0; i < 100; i++)
                tab[i]=i * i;
}
```

Il est possible d'utiliser des macro définitions dans d'autres macro définitions

Exemple ".c" :

```
#define LIG             12
#define COL             10
#define SURFACE         LIG * COL
#define CARRE(n)        n * n
#define CUBE(x)         n*CARRE(n)

int main(void)
{
        int tab[SURFACE];
        int i;

        for (i=0; i < SURFACE; i++)
                tab[i]=CUBE(i);
}
```

Code généré dans le ".i" :

```
int main(void)
{
        int tab[12 * 10];
        int i;

        for (i=0; i < 12 * 10; i++)
                tab[i]=i * i * i;
}
```

L'avantage des macro définitions est que si une des constantes doit changer pour évolution, le programmeur n'aura qu'à modifier la macro définition qui y est attribuée et à recompiler son programme. Il n'aura ainsi qu'une seule modification à apporter à un seul endroit du source et cette modification sera répercutée dans tout le code lors de la recompilation.

La directive "***#undef***" permet de supprimer les pseudo constantes ou pseudo fonctions crées quelques lignes plus haut. Si la macro définition n'existe pas l'instruction est simplement ignorée du pré processeur.

b) Les dangers - Les précautions à prendre

Le principal danger de la directive "***#define***" est l'utilisation des pseudo fonctions dans des calculs et le résultat de leurs traductions face aux priorités des opérateurs.

Exemple 1 :

```
#define CARRE(n)        n * n

i=CARRE(2 + 3)
// Vous aimeriez avoir "i=25"…
```

Traduction 1 :

```
i=2 + 3 * 2 + 3
// … mais vous n'avez que "i=11"
```

La solution pour éviter ce problème est d'encadrer chaque pseudo paramètre par des parenthèses "**(…)**"

Solution 1 :

```
#define CARRE(n)        (n) * (n)

i=CARRE(2 + 3)
// Vous aimeriez avoir "i=25"…
```

Résultat 1 :

```
i=(2 + 3) * (2 + 3)
// … et vous avez bien "i=25"
```

Les priorités sont aussi appliquées sur la pseudo fonction et pas uniquement dans les paramètres

Exemple 2 :

```
#define SOMME(x, y)     (x) + (y)

i=2 * SOMME(1, 2)
// Vous aimeriez avoir "i=6"…
```

Traduction 2 :

```
i=2 * (1) + (2)
// … mais vous n'avez que "i=4"
```

La solution pour éviter ce problème est d'encadrer la pseudo fonction dans sa globalité par des parenthèses "**(…)**" sans oublier les parenthèses sur les pseudo paramètres

Solution 2 :

```
#define SOMME(x, y)     ((x) + (y))

i=2 * SOMME(1, 2)
// Vous aimeriez avoir "i=6"…
```

Résultat 2 :

```
i=2 * ((1) + (2))
// … et vous avez bien "i=6"
```

Le troisième danger se nomme "**effet de bord**" et vient des opérateurs "++" et "--" qui agissent **directement** sur la variable qui les utilise sans qu'il y ait besoin d'expliciter l'affectation.

Exemple 3 :

```
#define CARRE(n)        ((n) * (n))

i=5
j=CARRE(i++)
// Vous aimeriez "i=6" et "j=25"…
```

Traduction 3 :

```
i=5
j=((i++) * (i++))
// … mais vous avez "i=7" et "j=30"
```

Il n'y a malheureusement **aucune** solution pour éliminer ce danger. Il ne reste donc qu'à **l'éviter** en ne mettant pas d'opérateur "++" ou "--" dans les arguments passés aux pseudo fonctions car vous ne savez pas forcément le traitement qu'ils vont subir. Ceci implique que vous **sachiez** que vous êtes en train d'utiliser une pseudo fonction ; c'est pourquoi les pseudo fonction, et par extension toutes les macro définitions, sont conventionnellement écrites en **majuscules**. Et afin d'avoir une certaine esthétique dans les macro définitions, il est aussi d'usage d'écrire les pseudo constantes en majuscules et d'y mettre des parenthèses.

c) Les macro définitions internes

Le pré processeur possède 4 macro définitions internes utilisables par le programmeur s'il le désire :

- __LINE__ : Constante décimale non-signée désignant le numéro de la ligne sur laquelle elle se trouve
- __FILE__ : Chaîne désignant le nom du fichier sur lequel le pré processeur est en train de travailler
- __DATE__ : Chaîne de la forme "Mmm jj aaaa" contenant la date de transformation du ".c" en ".i"
- __TIME__ : Chaîne de la forme "hh :mm :ss" contenant l'heure de transformation du ".c" en ".i"

Exemple :

```
int main(void)
{
        printf("Ce programme a été compilé le %s à %s – On est à la ligne %u du fichier %s\n",
__DATE__, __TIME__, __LINE__, __FILE__);
}
```

d) Création et suppression de macro définitions lors de la compilation

L'option "-Dmacro[=valeur]" de la commande "cc" permet de créer une macro définition lors de la compilation sans qu'elle soit forcément définie dans le fichier source.
Exemple de source :

```
int main(void)
{
        int tab[TAILLE];
        // La macro définition "TAILLE" n'existe pas
}
```

Exemple de la commande de compilation associée :

```
cc –DTAILLE=50 prog.c –o prog
```

L'option "-Umacro" de la commande "cc" permet de supprimer une macro définition lors de la compilation même si elle est définie dans le fichier source.

2) La compilation conditionnelle

Le pré processeur offre la possibilité d'établir une condition qui, si elle n'est pas respectée, aura pour résultat de ne pas traduire les lignes de code soumises à la condition. Ces dernières ne seront donc pas traitées par le compilateur.
La directive "***#if***" établi la condition de compilation. La directive facultative "***#elif***" permet d'établir une nouvelle condition si la précédente n'est pas vérifiée et la directive elle aussi facultative "***#else***" établi un branchement à traiter si aucune des conditions demandées n'est vérifiée. La directive obligatoire "***#endif***" termine le bloc de compilation conditionnelle. Les conditions peuvent s'appliquer sur la valeur des macro définitions et sur la valeur d'initialisation des variables.

Exemple :

```
#define DEBUG           (1)                         // Flag de débugging

int main(void)
{
        // Déclaration des variables
        int debug=1;                                // Variable de débugging

        // Programmation
#if DEBUG == 1                                      // Si la macro vaut "1"
        printf("Entrée dans le "main"\n");          // Affichage message de débugging
#elif debug == 1                                    // Si la variable a été initialisée à 1
        printf("Entrée dans le "main"\n");          // Affichage message de débugging
#else // DEBUG != 1 && debug != 1                   // Si rien n'a été vérifié
        printf("On n'affiche rien\n");              // On fait autre chose
#endif // DEBUG == 1                                // Fin de la compilation conditionnelle
        suite du code…
}
```

La directive "***#ifdef***" ou "***#ifndef***" permet de contrôler l'existence ou la non-existence d'une macro définition sans présumer de sa valeur. Ces deux directives peuvent également être suivies de la directive facultative "***#else***" mais se terminent obligatoirement par la directive "***#endif***".

Exemple :

```
#define DEBUG           (1)                         // Flag de débugging

int main(void)
{
        // Déclaration des variables
        …                                           // Variables diverses

        // Programmation
#ifdef DEBUG                                        // Si la macro est définie
        printf("Entrée dans le "main"\n");          // Affichage message de débugging
#else // DEBUG                                      // Si la macro n'est pas définie
        printf("On n'affiche rien\n");              // On fait autre chose (ou on ne fait rien)
#endif // DEBUG                                     // Fin de la compilation conditionnelle
        suite du code…
}
```

3) L'inclusion des fichiers

a) Utilisation

La directive "***#include***" insère à l'endroit où elle se trouve un fichier qui sera traité comme s'il faisait partie du source. Cette directive peut s'écrire de deux façons différentes :

☞ #include <fichier> (avec les signes "**<**" et "**>**") : le pré processeur va chercher le fichier demandé dans le répertoire "/usr/include" et il est possible de lui rajouter d'autres répertoires de recherche avec l'option "-I rep" de la commande "cc".

☞ #include "fichier" (avec les guillemets **"**) : on est obligé d'indiquer le nom exact (relatif ou absolu) du fichier dans la machine.

En général, ces fichiers étant inclus en début de source, on les appelle des "**headers**" (fichiers d'en-tête) et sont par convention nommés "xxx**.h**". Ils contiennent en général des macro définitions, des prototypes et des nouveaux types ; et leur but est de faciliter la maintenance d'un projet où tous les sources incluant le même fichier seront facilement modifiables si les conditions changent (il suffira de modifier le fichier inclus et de recompiler tous les sources).

Remarque :
Il est possible, si on désire écrire une fonction pour les autres programmeurs, d'écrire le code de cette fonction dans un fichier ".c" et inclure ce fichier dans les autres programmes. Mais cette méthode est contraire à l'éthique du C car elle ne garanti plus la robustesse du code (l'auteur de la fonction n'est pas certain que celle-ci n'est pas modifiée dans son utilisation). Il vaut mieux pour cela écrire le code de la fonction dans un source, en faire un module objet ".o" ou une librairie ".a" ; écrire ensuite le prototype de la fonction dans un en-tête ".h" et offrir aux autres programmeurs l'en-tête et le module objet ou la librairie qui va avec. Ainsi, personne ne pourra modifier le code de la fonction.

Exemple de début de programme "prog.c" :

```
#include <fichier1.h>                          // Sera pris dans "/usr/include"
#include "fichier2.h"                          // Sera pris dans le répertoire courant "."
#include "../include/fichier3.h"               // Sera pris dans "../include"
#include "/home/prog/include/fichier4.h"       // Sera pris dans "/home/prog/include"
…suite du code
```

Exemple de compilation du programme "prog.c" :

```
cc –I/home/local/include –I../include prog.c –o prog
Le fichier "fichier1.h" sera aussi cherché dans "/home/local/include" et dans "../include"
```

Le fichier le plus utilisé couremment est "stdio.h" (standards entrées-sorties) car il donne le prototype des fonctions d'entrées-sorties comme "printf" ou "scanf". De plus, il définit certaines macro définitions comme le pointeur "NULL" qui vaut "(void*)0" ; c'est à dire "zéro transformé en pointeur universel" .

b) Les dangers - Les précautions à prendre

Le principal danger de l'inclusion de fichiers est d'inclure un fichier "x" et un fichier "y" sans savoir que "x" inclut lui aussi "y" ("x" a aussi le droit d'avoir des directives "#include"). Ce danger se nomme "***idempotence***". Cela peut provoquer de simples avertissements lors de la compilation si la même macro définition est vue deux fois par le compilateur mais aussi des erreurs de compilation si le même prototype de fonction est vu deux fois.
Pour se prévenir de ce danger, il est nécessaire d'utiliser la compilation conditionnelle en utilisant une macro définition en sentinelle. En général, la macro définition porte le nom du fichier dans lequel elle se trouve.

Exemple d'un fichier "fichier1.h" :

```
#ifndef _FICHIER1_H_                    // Test sur la non-existence de cette macro définition
#define _FICHIER1_H_                    // Création de la macro définition
… définition de tous les prototypes et macro contenues dans le fichier 1…

#include <fichier2.h>                   // Inclusion fichier2
#endif // _FICHIER1_H_                  // Fin du test concernant cette macro définition
```

Exemple d'un fichier "fichier2.h" :

```
#ifndef _FICHIER2_H_                    // Test sur la non-existence de cette macro définition
#define _FICHIER2_H_                    // Création de la macro définition
… définition de tous les prototypes et macro contenues dans le fichier 2…
#endif // _FICHIER2_H_                  // Fin du test concernant cette macro définition
```

Exemple de début de programme "prog.c" :

```
#include <fichier1.h>                   // Demande d'inclusion de "fichier1.h"
#include <fichier2.h>                   // Demande d'inclusion de "fichier2.h"
                                        //      Le programmeur ne sait pas que
                                              "fichier1.h" inclut déjà "fichier2.h"    */
…suite du code
```

Résultat de la traduction par le pré processeur :

```
// Première ligne du programme : "#include <fichier1.h>
#ifndef _FICHIER1_H_                    // Non-existence de cette macro définition => vrai
#define _FICHIER1_H_                    // Création de la macro définition
…définition de tous les prototypes et macro contenues dans le fichier 1…
        // Inclusion de "fichier2"
        #ifndef _FICHIER2_H_            // Non-existence de cette macro définition => vrai
        #define _FICHIER2_H_            // Création de la macro définition
        … définition de tous les prototypes et macro contenues dans le fichier 2…
        #endif // _FICHIER2_H_
#endif // _FICHIER1_H_

// Seconde ligne du programme : "#include <fichier2.h>
#ifndef _FICHIER2_H_                    // Non-existence de cette macro définition => faux
#define _FICHIER2_H_                    // Ce code ne sera pas retraité
…pas de redéfinition des prototypes et macro déjà définies et donc déjà connues…
#endif // _FICHIER2_H_

…suite du code
```

4) La gestion des chaînes

L'opérateur "#" remplace la chaîne qui le suit par une chaîne entre guillemets jusqu'au premier espace rencontré. Cela permet de traiter comme chaîne des éléments tels que des nombres.

Exemple :

```
#define CHAINE(x)               #x
int main(void)
{
        printf("La longueur du nombre 12 est %d\n", strlen(#12));
}
```

L'opérateur "##" forme une chaîne de caractères à partir de plusieurs termes (concaténation). Le mot obtenu peut être ensuite utilisé comme identificateur.
Exemple :

```
#define INDICE(x)               var##x
int main(void)
{
        int var1, var2, var3;
        int i;
        for (i=1; i <= 3; i++)
                var##i=0;                       // "var##i" sera remplacé par var1, var2, var3
}
```

5) Les autres directives

La directive "***#error message***" demande au compilateur d'arrêter la compilation en affichant le message comme s'il y avait une erreur dans le programme source. C'est utilisé surtout avec la compilation conditionnelle si la non-vérification de la condition rend inutile de poursuivre la compilation.
Exemple :

```
#ifndef _UNIX                                   // Si "_UNIX" n'est pas défini
#ifndef _LINUX                                  // Si "_LINUX" n'est pas défini
#error Ce source ne fonctionne que sur Unix ou Linux
#endif // _LINUX                                // Fin du test sur "_LINUX"
#endif // _UNIX                                 // Fin du test sur "_UNIX
… suite du code
```

La directive "***#pragma***" permet de donner des ordres spécifiques au pré processeur. Son utilisation est dépendante du pré processeur utilisé et on doit se référer à la documentation spécifique de chaque compilateur pour connaître les directives possibles.

XIII) LES PARAMETRES DE "MAIN()"

1) Explication

main() est une fonction comme une autre. Elle reçoit donc des paramètres et renvoie une valeur.

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

Explication des paramètres :

- int argc (vient de "*argument count*") : Nombre d'arguments passés au programme ; y compris le nom du programme lui-même. Il s'ensuit que cette variable ne peut jamais être à zéro.
Rappel: Les arguments passés à un programme sont des éléments sur lequel il doit travailler (ex : cp fic1 fic2 => "cp" est le programme, "fic1" est son 1er argument, "fic2" est son second argument)

- char *argv[](vient de "*argument value*") : Pointeur vers un tableau de pointeurs, chacun de ceux-ci pointant vers un tableau de caractères au format ***chaîne*** (tableau contenant plusieurs caractères dont le dernier de valeur zéro). Chaque chaîne contiendra le nom de l'argument passé au programme. De plus, "***argv[0]***" contiendra le nom du programme lui-même qui est considéré comme le premier des arguments. Enfin, le tableau "**argv**" contiendra un pointeur supplémentaire de valeur nulle. On remarquera que cette valeur nulle se trouve dans la variable "***argv[argc]***". Cette valeur nulle peut donc servir de condition d'arrêt dans une boucle à la place de "***argc***".

- char *envp[](vient de "*environment program*"). De même format que "argv", cette variable pointe vers un tableau de chaînes de caractères. Chaque chaîne contiendra la valeur de chaque variable d'environnement qui a été exportée par un des processus ascendants (dans le monde UNIX) et donc accessible au programme. Cette chaîne est de la forme "***variable=valeur***". De même que pour "argv", un dernier pointeur de ce tableau a pour valeur "zéro". Il est heureux que ce soit le cas car sinon, le programmeur n'aurait aucun moyen de s'arrêter dans le cas d'un balayage des variables d'environnement.

Valeur renvoyée (int) : un entier utilisable par le processus appelant le programme (récupéré par la variable "***$?***"en shell Unix).

2) Schéma

Exemple d'appel d'un programme "prog" avec 3 arguments :

```
prog toto titi tata
```

Image de la mémoire à ce moment là :

Adresse	Nom	Contenu
0x1000	argc	4 (nb arguments + nom du programme)
0x1… etc.	…	*variables quelconques…*
0x2000	argv	0x5000
0x2… etc.	…	*variables quelconques…*
0x5000	argv[0]	0x6000
0x5004	argv[1]	0x7000
0x5008	argv[2]	0x9000
0x500C	argv[3]	0xA000
0x5010	argv[4]	0 (fin des arguments)
0x5… etc.	…	*variables quelconques…*
0x6000	argv[0][0]	'p'
0x6001	argv[0][1]	'r'
0x6002	argv[0][2]	'o'
0x6003	argv[0][3]	'g'
0x6004	argv[0][4]	'\0'
0x6… etc.	…	*variables quelconques…*
0x7000	argv[1][0]	't'
0x7001	argv[1][1]	'o'
0x7002	argv[1][2]	't'
0x7003	argv[1][3]	'o'
0x7004	argv[1][4]	'\0'
0x7… etc.	…	*variables quelconques…*
0x9000	argv[2][0]	't'
0x9001	argv[2][1]	'i'
0x9002	argv[2][2]	't'
0x9003	argv[2][3]	'i'
0x9004	argv[2][4]	'\0'
0x9… etc.	…	*variables quelconques…*
0xA000	argv[3][0]	't'
0xA001	argv[3][1]	'a'
0xA002	argv[3][2]	't'
0xA003	argv[3][3]	'a'
0xA004	argv[3][4]	'\0'
0xA… etc.	…	*variables quelconques…*

3) Exemple

Exemple permettant de traiter les arguments :

```
// Fonction principale du programme
int main(
        int argc,                       // Nombre d'arguments passés au programme
        char *argv[],                   // Liste des arguments passés au programme
        char *envp[])                   // Environnement de travail du programme
{
        // Déclaration des variables
        int i;                          // Indice de boucle sur "argv" ou "envp"
        char **pt;                      // Pointeur de boucle sur "argv" ou "envp"

        // Affichage du nom du programme et du nombre d'arguments reçus
        printf("Le programme %s a reçu %u arguments\n", argv[0], argc);

        // Si ce programme a reçu des arguments (en dehors de son nom)
        if (argc > 1)
        {
                // On va afficher les arguments
                printf("Les arguments pasés au programme sont: ");

                // Une boucle sur le nombre d'arguments
                for (i=1; i < argc; i++)
                        printf("\tArgument %u/%u: <%s>\n", i + 1, argc, argv[i]);

                // La même boucle sur "argv" mais en utilisant un pointeur (boucle plus rapide)
                for (i=1, pt=argv; i < argc; i++, pt++)
                        printf("\tArgument %u/%u: <%s>\n", i , argc, *pt);
}
        else
                // Il n'y a aucun argument et on le dit
                printf("En dehors de son nom <%s>, ce programme n'a aucun argument\n", argv[0]);

        // On va afficher l'environnement du programme
        printf("L'environnement que peut récupérer le programme est: ");

        // Une boucle sur envp[i] jusqu'à trouver "0"
        for (i=0; envp[i] != NULL; envp++)
                printf("\tVariable %u: <%s>\n", i + 1, envp[i]);

        // La même boucle sur "envp" mais en utilisant un pointeur (boucle plus rapide)
        for (pt=envp, i=0; *pt != NULL; pt++, i++)
                printf("\tVariable %u: <%s>\n", i + 1, *pt);

        // Fin du programme
        return 0;
}
```

XIV) LA BIBLIOTHEQUE STANDARD

Ce chapitre va faire un rapide descriptif des fonctions les plus usitées offertes aux programmeurs et disponibles dans de la librairie standard "**/usr/lib/libc.a**" automatiquement liée par le compilateur Unix "**cc**" ou "**gcc**"

1) Gestion de la mémoire

a) Fonctions de réservation et de libération de zones mémoires

Les fonctions ***malloc()***, ***calloc()*** et ***realloc()*** permettent respectivement d'allouer une zone mémoire de façon dynamique (quand la taille à allouer n'est pas connue du programmeur), allouer la zone mémoire en initialisant tous ses octets à zéro, et réallouer une zone mémoire avec une taille différente (le plus souvent supérieure) de l'allocation précédente. Ces fonctions servent lorsqu'on veut créer des tableaux dont on ne connaît pas la taille à l'avance.
Cette allocation réserve une zone en mémoire principale du programme. Donc, même si l'allocation se fait à l'intèrieur d'une fonction, la zone allouée sera disponible dans tout le reste du programme pour peu que celui-ci connaisse l'adresse de la zone allouée (cette adresse peut, par exemple, être renvoyée par la fonction qui réalise l'allocation).

```
#include <sys/types.h>                    // Création de nouveaux types
#include <stdlib.h>                       // Utilisation des librairies standard
void *malloc (size_t Gsize);
void *calloc (size_t nb, size_t Esize);
void *realloc (void *ptr, size_t Gsize);
```

Explication des paramètres :

- size_t Gsize : La taille globale en octets de la mémoire demandée
- size_t nb : Le nombre d'éléments demandés
- size_t Esize : La taille en octets d'un élément de la zone
- void *ptr : Pointeur sur la zone à réallouer

Valeur renvoyée (void *) :

- Pointeur sur la zone allouée si la mémoire est disponible
- NULL si une erreur s'est produite ou s'il n'y a pas assez de mémoire disponible

La fonction ***free()*** libère une zone mémoire allouée par "*malloc()*", "*calloc()*" ou "*realloc()*". Il est **impératif** de libérer toutes les zones allouées quand elles deviennent inutiles, surtout pour un programme tournant sous *Windows* car dans le cas contraire, la zone allouée n'est plus jamais libérée (même quand le programme s'arrête) et il est alors nécessaire de redémarrer l'ordinateur si on veut récupérer la mémoire perdue par l'allocation.

```
#include <stdlib.h>                       // Utilisation des librairies standard
void free (void *ptr);
```

Explication des paramètres :

- void *ptr : pointeur sur la zone à libérer

b) Fonctions de manipulation de zones mémoires

La fonction ***memcmp()*** permet de comparer deux zones mémoires. La comparaison se fera octet par octet jusqu'à ce qu'il y ait une différence ou jusqu'à ce que le nombre d'octets à comparer soit atteint.

```
#include <memory.h>                                    // Manipulations mémoires
int memcmp (void *mem1, char *mem2, size_t n);
```

Explication des paramètres :

- void *mem et void *mem2 : pointeur sur les zones à comparer
- size_t n : Le nombre d'octets sur lesquels se fera la comparaison

Valeur renvoyée (int) : Ces fonctions renvoient trois types de valeur :

- Une valeur inférieure à zéro si "mem1" est plus petit que "mem2"
- Une valeur égale à zéro si "mem1" est égal à "mem2"
- Une valeur supérieure à zéro si "mem1" est plus grand que "mem2"

Les fonctions ***memset()*** et ***memcpy()*** permettent respectivement d'initialiser chaque octet d'une zone mémoire avec une valeur spécifique et de copier une zone mémoire dans une autre.

```
#include <memory.h>                                    // Manipulations mémoires
void *memset (void *ptr, int val, size_t n);
void *memcpy (void *dest, void *src, size_t n);
```

Explication des paramètres :

- void *ptr : Pointeur sur la zone mémoire à remplir
- int val : Valeur à mettre dans chaque octet de la zone mémoire
- void *dest : Pointeur sur la zone mémoire de destination
- void *src : Pointeur sur la zone mémoire source
- size_t n : Nombre d'octets à traiter

Valeur renvoyée (void*) : Ces fonctions renvoient un pointeur sur la zone mémoire modifiée (ptr ou dest).

La fonctions ***memchr()*** permet de rechercher une valeur codée sur un octet (caractère) dans une zone mémoire

```
#include <memory.h>                                    // Manipulations mémoires
void *memchr (void *ptr, int val, size_t n);
```

Explication des paramètres :

- void *ptr : Pointeur sur la zone mémoire à examiner
- int val : Valeur à chercher
- size_t n : Nombre d'octets à traiter

Valeur renvoyée (char*) :

- pointeur sur la zone où se trouve le caractère cherché s'il est présent
- NULL si le caractère cherché ne s'y trouve pas

c) Exemple

Faire un programme qui permette d'enregistrer dans un tableau autant de nombres que l'utilisateur voudra. Tant que l'utilisateur saisi un nombre, le programme le stocke et gère son tableau de stockage pour pouvoir toujours avoir suffisemment de place.

```
#include <stdio.h>                              // I/O bufferisées

#define SZ_ALLOC                (10)            // Taille allocation mémoire

// Déclaration des structures utiles
typedef struct {
        float *elem;                            // Tableau de stockage des nombres
        float *ptr;                             // Ptr de balayage
        unsigned long nb;                       // Nb éléments du tableau
        unsigned long size;                     // Taille allouée au tableau
        unsigned long ind;                      // Indice de balayage
} t_tableau;                                    // Type global du tableau

// Fonction principale du programme
int main(void)
{
        // Déclaration des variables
        float saisie;                           // Nombre saisi
        t_tableau tabNB;                        // Tableau de stockage dynamique

        // Initialisation
        tabNB.nb=0;
        tabNB.size=SZ_ALLOC;
        if ((tabNB.elem=(float*)malloc(tabNB.size * sizeof(float))) == NULL)
        {
                // L'allocation a échoué – On quitte le programme
                printf("Erreur allocation %lu\n", tabNB.size * sizeof(float));
                exit(1);
        }

        // Boucle de saisie
        while(1)
        {
                printf("Entrez le nombre à stocker (0 pour quitter) :");
                scanf("%f, &saisie);
                if (saisie == 0.0)
                        break;                  // Sortie de la saisie

                // On vérifie si l'indice du tableau a atteint sa taille allouée
                if (tabNB.nb == tabNB.size)
                {
                        // On agrandit la taille allouée de "n" éléments
                        tabNB.size+=SZ_ALLOC;

                        // On réalloue le tableau et on vérifie l'allocation
                        if ((tabNB.elem=(float*)realloc(tabNB.elem, tabNB.size * sizeof(float))) == NULL)
                        {
                                // La réallocation a échoué – On quitte le programme
                                printf("Erreur réallocation %lu\n", tabNB.size * sizeof(float));
                                exit(1);
                        }
                }

                // On stoce le nombre saisi dans sa place qui est disponible
                tabNB.elem[tabNB.nb]=saisie;

                // On prépare l'indice pour la saisie suivante
```

```
                tabNB.nb++;
        }

        // Récapitulatif du contenu de "tabNB" à ce point du programme
                elem contient l'adresse du début du tableau
                nb contient son nombre d'éléments
                size contient sa taille allouée
                ind et ptr ne sont pas encore utilisés
        */

        // On affiche les nombres saisis
        for (tabNB.ind=0, tabNB.ptr=tabNB.elem; tabNB.ind < tabNB.nb; tabNB.ind++, tabNB.ptr++)
                printf("Le nombre %lu vaut %f\n", tabNB.ind + 1, *tabNB.ptr);

        // On n'a plus besoin du tableau – Le programme est fini
        if (tabNB.elem != NULL)
                free(tabNB.elem);

        // Fin du programme
        return 0;
}
```

2) Gestion des caractères

Ces fonctions ont pour but de manipuler les caractères

a) Fonctions de vérification de la catégorie du caractère

Liste des fonctions de vérification en vrac :

```
#include <ctype.h>                  // Gestion des caractères
int isalnum (char c);               // Vrai si "c" est alphanumérique, lettre ou chiffre
int isalpha (char c);               // Vrai si "c" est alphabétique (sans accent)
int isascii (char c);               // Vrai si "c" est au standard ASCII (entre 0 et 127)
int isblank (char c);               // Vrai si "c" est blanc (espace ou tabulation)
int iscntrl (char c);               // Vrai si "c" est caractère de contrôle (entre 1 et 26)
int isdigit (char c);               // Vrai si "c" est un chiffre décimal
int isgraph (char c);               // Vrai si "c" est imprimable non "blanc"
int islower (char c);               // Vrai si "c" est une minuscule (sans accent)
int isprint (char c);               // Vrai si "c" est imprimable ("blanc" compris)
int ispunct (char c);               // Vrai si "c" est caractère de ponctuation
int isspace (char c);               // Vrai si "c" est caractère d'espacement
int isupper (char c);               // Vrai si "c" est une majuscule
int isxdigit (char c);              // Vrai si "c" est un chiffre hexadécimal
```

b) Fonctions de modification de la catégorie de caractère

Liste des fonctions de modification en vrac :

```
#include <ctype.h>                  // Gestion des caractères
int toascii (char c);               // Transforme "c" en ASCII standard (entre 0 et 127)
int toupper (char c);               // Transforme "c" en majuscule (accent conservé)
int tolower (char c);               // Transforme "c" en minuscule
```

Remarque : Malgré le type ***int*** de ces fonctions ; elles renvoient bien un caractère ; c'est à dire un nombre compris entre 0 et 128.

c) Dangers

Dans beaucoup de vieux systèmes, ces fonctions décrites comme telles ne sont en fait que des macro définitions (preuve que les conventions ne sont pas toujours respectées, même par ceux qui les ont instaurées) utilisant plusieurs fois le caractère passé en paramètre ; ce qui peut donc générer des erreurs difficilement décelables si on les utilise avec les opérateurs "**++**" ou "**--**" (effets de bord).

3) Gestion des chaînes de caractères

Toutes ces fonctions obéissent aux règles suivantes :

- Chaque tableau de caractères qui est passé à la fonction doit contenir la valeur **'\0'** permettant de repérer la fin du tableau
- Chaque tableau de caractères écrit par la fonction contiendra la valeur **'\0'** stocké par cette dernière (sauf dans les cas particulier des fonctions "***strncpy()***" et "***strndup()***")
- Chaque pointeur éventuellement renvoyé par la fonction pointera sur un tableau de caractères contenant aussi la valeur **'\0'** positionnée par la fonction

Ces règles sont la base des techniques de manipulation de chaînes de caractères en langage C (n'oubliez pas qu'une chaîne de caractères n'est qu'un tableau de caractères délimité par la valeur *'\0'*).

La fonction ***strlen()*** permet de compter la longueur d'une chaîne de caractères. La fonction cherche et calcule la position du *'\0'* dans le tableau de caractères reçu en argument et renvoie cette position (d'où l'utilité que le programmeur soit certain que ce *'\0'* soit bien présent dans le tableau envoyé à la fonction).
Il est important de faire la distinction entre l'opérateur ***sizeof()*** qui renvoie la taille globale d'une zone mémoire et la fonction ***strlen()*** qui indique combien on trouve de caractères situés avant le *'\0'* terminant la chaîne. Sur un "*char chaine[100]*", l'opérateur ***sizeof()*** renverra *100* alors que la fonction ***strlen()*** renverra le nombre de caractères situés avant le caractère spécial '\0'.

```
#include <string.h>                                    // Gestion des chaînes de caractères
int strlen (char *ch);
```

Explication des paramètres :

- char *ch : La chaîne dont on désire connaître la longueur

Valeur renvoyée (int) : La longueur de la chaîne

Les fonctions ***strcmp()*** et ***strncmp()*** permettent de comparer deux chaînes de caractères. La comparaison se fera octet par octet jusqu'à ce qu'il y ait une différence et la fonction ***strncmp()*** ne fera la comparaison que sur les "n" premiers octets.

```
#include <string.h>                                    // Gestion des chaînes de caractères
int strcmp (char *ch1, char *ch2);
int strncmp (char *ch1, char *ch2, size_t n);
```

Explication des paramètres :

- char *ch1 et char *ch2 : Les deux chaînes à comparer
- size_t n : le nombre de caractères sur lesquels se fera la comparaison

Valeur renvoyée (int) : Ces fonctions renvoient trois types de valeur :

- Une valeur inférieure à zéro si "ch1" est plus petit que "ch2"
- Une valeur égale à zéro si "ch1" est égal à "ch2"
- Une valeur supérieure à zéro si "ch1" est plus grand que "ch2"

Les fonctions ***strcpy()***, ***strncpy()***, ***strcat()*** et ***strncat()*** permettent respectivement de copier une chaîne dans une autre, copier seulement les "n" premiers caractères d'une chaîne dans une autre, concaténer une chaîne à la suite d'une autre et de concaténer seulement les "n" premiers caractères d'une chaîne à la suite d'une autre.
Les fonctions ***strdup()*** et ***strndup()*** permettent, tout comme comme le font ***strcpy()*** et ***strncpy()***, de copier une chaîne dans une autre ou de copier seulement les "n" premiers caractères d'une chaîne dans une autre ; cependant, elles assurent en plus l'allocation mémoire suffisante pour recevoir la chaîne d'origine, ce que ne font pas les fonctions ***strcpy()*** et ***strncpy()***. Il s'agit en fait d'un mixage des fonctions ***malloc()*** et ***strcpy()***. Bien entendu, il est nécessaire ; en fin d'utilisation des zones mémoires créées ; de les libérer par un appel à la fonction ***free()***.
Attention: Les fonctions ***strncpy()*** et ***strndup()*** ne mettront le '\0' final dans la chaîne de destination que si elles ont copié toute la chaîne source. Inversement si la chaîne source fait plus des "n" caractères demandés pour la copie, les fonctions ne rajouteront pas le '\0' dans la chaîne de destination.

```
#include <string.h>                                    // Gestion des chaînes de caractères
char *strcpy (char *dest, char *src);
char *strncpy (char *dest, char *src, size_t n);
char *strcat (char *dest, char *src);
char *strncat (char *dest, char *src, size_t n);
char *strdup (char *src);
char *strndup (char *src, size_t n);
```

Explication des paramètres :

- char *dest : La chaîne de destination (qui doit avoir la place de recevoir ce que la fonction lui mettra plus le caractère '\0'
- char *src : La chaîne source (qui doit contenir le caractère '\0')
- size_t n : le nombre de caractères pris de la chaîne source

Valeur renvoyée (char*) : Ces fonctions renvoient un pointeur sur la chaîne de destination.

Les fonctions ***strchr()***, ***strrchr()*** et ***strstr()*** permettent respectivement de recherche un caractère dans une chaîne en partant du début ou en partant de la fin de la chaîne ; et de rechercher une sous-chaîne contenue dans une chaîne.

```
#include <string.h>                                    // Gestion des chaînes de caractères
char *strchr (char *ch, int carac);
char *strrchr (char *ch, int carac);
char *strstr (char *ch, char *sousch);
```

Explication des paramètres :

- char *ch : La chaîne de recherche (qui doit contenir le caractère '\0')
- int carac : Le caractère à rechercher (converti en entier mais cela n'a pas d'importance puisque l'entier a un codage plus grand que le caractère)
- char *sousch : La sous-chaîne à rechercher (qui doit contenir le caractère '\0')

Valeur renvoyée (char*) : Ces fonctions renvoient un pointeur sur le caractère trouvé ou un pointeur sur le début de la sous-chaîne trouvée.

Les fonctions ***strcspn()*** et ***strcspnr()*** renvoient le nombre de caractères d'une chaîne qui contient respectivement que des caractères d'une sous-chaîne ou aucun des caractères d'une sous-chaîne.

```
#include <string.h>                                    // Gestion des chaînes de caractères
size_t strspn (char *ch, char *accept);
size_t strcspn (char *ch, char *reject);
```

Explication des paramètres :

- char *ch : La chaîne de recherche (qui doit contenir le caractère '\0')
- char *accept : La sous-chaîne des caractères acceptés (qui doit contenir le caractère '\0')
- char *reject : La sous-chaîne des caractères rejetés (qui doit contenir le caractère '\0')

Valeur renvoyée (size_t) : Ces fonctions renvoient le nombre de caractères débutant la chaîne et correspondants à la demande.

La fonction ***strpbrk()*** recherche la première occurrence d'un des caractères de la sous-chaîne dans une chaîne.

```
#include <string.h>                                    // Gestion des chaînes de caractères
char *strpbrk (char *ch, char *sousch);
```

Explication des paramètres :

- char *ch : La chaîne de recherche (qui doit contenir le caractère '\0')
- char *sousch : La sous-chaîne des caractères cherchés (qui doit contenir le caractère '\0')

Valeur renvoyée (char *) : La fonction renvoie un pointeur sur le premier caractère de la sous-chaîne trouvé dans la chaîne.

La fonction ***fnmatch()*** permet de vérifier si une chaîne correspond à un motif pouvant contenir des méta caractères shell.

```
#include <fnmatch.h>                                   // Gestion des motifs ressemblants
int fnmatch (char *motif, char *ch, int flag);
```

Explication des paramètres :

- char *motif : Le motif de recherche (qui doit contenir le caractère '\0')
- char *ch : La chaîne à vérifier (qui doit contenir le caractère '\0')
- int flag : Flag modifiant le comportement de la fonction. Celui-ci peut être composé au moyen d'un "**ou bit à bit**" d'une ou plusieurs des trois constantes suivantes :
  - FNM_PATHNAME : les caractères slashes ("/") ne sont pas mis en comparaison avec le motif (utilisé si on cherche à mettre en correspondance des noms de fichiers).
  - FNM_PERIOD : Le caractère point (".") ; s'il se trouve en début de la chaîne ; ne sera pas mis en comparaison avec le motif (utilisé aussi pour les noms de fichiers).
  - FNM_QUOTE : Les quotes ou doubles-guillemets ("'"") ne seront pas mis en comparaison avec le motif (permet d'utiliser des méta caractères en tant que caractères simples).

Les fonctions ***atoi()***, ***atol()*** et ***atof()*** convertissent une chaîne de caractères représentant un nombre (ex "128") en nombre correspondant. Chaque fonction renverra respectivement un nombre de type "int", un nombre de type "long" ou un nombre de type "double". Lors de la conversion, chaque fonction s'arrête dès qu'elle rencontre un caractère qu'elle ne sait pas traduire en nombre.

```
#include <stdlib.h>                                    // Fonctions de la librairie standard
int atoi (char *ch);
long atol (char *ch);
double atof (char *ch);
```

Explication des paramètres :

- char *ch : La chaîne contenant le nombre à convertir (qui doit contenir le caractère '\0')

Valeur renvoyée (int, long ou double) : Le nombre correspondant à la chaîne dans la précision de la fonction utilisée.

Les fonctions ***strtol()***, ***strtoul()*** et ***strtod()*** convertissent une chaîne de caractères représentant un nombre dans une base quelconque (ex "0x128") en nombre correspondant dans la base désirée. Chaque fonction renverra respectivement un nombre de type "long", un nombre de type "unsigned long" ou un nombre de type "double". Lors de la conversion, chaque fonction s'arrête dès qu'elle rencontre un caractère qu'elle ne sait pas traduire et il est possible de lui faire remplir une zone avec les erreurs qu'elle a pu rencontrer.

```
#include <stdlib.h>                                    // Fonctions de la librairie standard
long strtol (char *ch, char **err, int base);
unsigned long strtoul (char *ch, char **err, int base);
double strtod (char *ch, char **err);
```

Explication des paramètres :

- char *ch : La chaîne contenant le nombre à convertir (qui doit contenir le caractère '\0')
- char **err : La zone destinée à recevoir les erreurs rencontrées. Il est possible de ne pas utiliser ce paramètre en y mettant la constante "NULL"
- int base : base de conversion. Celle-ci peut-être comprise entre 2 et 32. Passer la valeur "0" demande à la fonction de convertir en base "10" par défaut.

Valeur renvoyée (int, long ou double) : Le nombre correspondant à la chaîne dans la base demandée

Les fonctions ***sscanf()*** et ***sprintf()*** ont pour but d'aller respectivement extraire des valeurs et les stocker dans des variables à partir d'une chaîne ou d'une variable déjà existante et écrire des chaînes formatées dans une variable créé pour l'occasion.

```
#include <stdio.h>                                     // I/O bufferisées
int sscanf (char *pt, …);
int sprintf (char *pt, …);
```

Explication des paramètres :

- char *pt : chaîne de caractères destinée à fournir des valeurs (***sscanf***) ou récupérer ce qui est écrit (***sprintf***).
- … format de saisie ou d'affichage correspondant aux fonctions "***scanf()***" et "***printf()***"

Valeur renvoyée (int) :

- Le nombre d'octets lus ou écrits dans la chaîne.
- 0 si rien n'a été fait (format de saisie non trouvé) ou si une erreur s'est produite

4) Gestion des fichiers mode buffer

a) Fonctions d'ouverture et de fermeture de fichiers

La fonction ***fopen()*** demande au système d'ouvrir un fichier normal afin d'y accéder (lire ou écrire). Cette fonction utilise en paramètre un nom de fichier et renvoie un pointeur sur un type "***FILE***" dont la structure est définie dans le fichier "***/usr/include/stdio.h***". Ce pointeur sera ensuite l'unique référence concernant le fichier qu'il faudra utiliser.
Toutes les opérations de lectures et d'écritures seront bufférisées, c'est à dire gardées en mémoire pour optimiser les accès disque.

```
#include <stdio.h>                                  // I/O bufferisées
FILE *fopen (char *nom, char *mode);
```

Explication des paramètres :

- char *nom : chaîne de caractères contenant le nom du fichier à ouvrir
- char *mode : chaîne de caractères contenant le mode d'ouverture demandé. Cette chaîne peut être :
  - "r" : le fichier est ouvert en lecture seule
  - "w" : le fichier est vidé puis ouvert en écriture
  - "a" : le fichier est ouvert pour écriture en fin de fichier
  - "r+" : le fichier est ouvert en lecture avec possibilité d'écriture
  - "w+" : le fichier est vidé puis ouvert en écriture avec possibilité de lecture
  - "a+" : le fichier est ouvert pour écriture en fin de fichier mais possibilité de lecture

Remarque : Dans le monde *Windows*, ces modes d'ouverture ne s'appliquent que pour des fichiers ascii ou "fichiers texte". Dans le cas d'une ouverture d'un fichier "binaire", chaque chaîne du mode doit inclure la lettre "b". Ce distingo n'existe pas dans le monde *Unix*.

Valeur renvoyée (FILE*) :

- pointeur du fichier ouvert. Ce pointeur servira de référence ultérieure chaque fois qu'il faudra accéder au fichier.
- Macro définition "NULL" si le fichier n'a pas pu être ouvert pour une raison quelconque

La fonction ***fclose()*** ferme un fichier ouvert préalablement par ***fopen()***. Si des données sont encore en attente d'écriture sur disque, le tampon est vidé et les données sont écrites physiquement sur le fichier disque.

```
#include <stdio.h>                                  // I/O bufferisées
int fclose (FILE *pt);
```

Explication des paramètres :

- FILE *pt : pointeur du fichier renvoyé par la fonction ***fopen()***

Valeur renvoyée (int) :

- 0 si toute l'opération se passe correctement
- Macro définition "EOF" dans le cas contraire

Remarque : Chaque programme en exécution possède intrinsèquement trois pointeurs sur un type *FILE* ouverts et fermés automatiquement par le système au démarrage et à la terminaison du programme (ne nécessitant donc pas d'appel à "*fopen*" ou "*fclose*") et associés aux périphériques de travail :

- stdin : tampon associé au clavier
- stdout : tampon associé aux affichages des messages normaux du programme
- stderr : tampon associé aux affichages des messages d'erreur du programme

b) Fonctions de lecture et d'écriture

Les fonctions ***fgetc()***, ***getc()***, ***fputc()*** et ***putc()*** ont pour but d'aller respectivement lire et écrire un octet dans un fichier préalablement ouvert avec la fonction ***fopen()***.
Les fonctions *getc()* et *putc()* diffèrent de leurs homologues par le fait que ce sont des macro définitions. Elles s'exécutent donc plus rapidement que leurs homologuent mais peuvent générer des effets de bord si elles sont mal utilisées.

```
#include <stdio.h>                                  // I/O bufferisées
int fgetc (FILE *pt);
int getc (FILE *pt);
int fputc (int carac, FILE *pt);
int putc (int carac, FILE *pt);
```

Explication des paramètres :

- int carac : caractère à écrire transformé par la fonction en "unsigned char"
- FILE *pt : pointeur du fichier renvoyé par la fonction ***fopen()***

Valeur renvoyée (int) :

- Caractère lu ou écrit par la fonction converti en "int"
- Macro définition "EOF"s'il n'y a plus rien à lire ou si une erreur s'est produite

Les fonctions ***getchar()*** et ***putchar()*** ont pour but d'aller respectivement lire et écrire un octet dans le tampon clavier ou écran.

```
#include <stdio.h>                                  // I/O bufferisées
int getchar (void);
int putchar (int carac);
```

Explication des paramètres :

- int carac : caractère à écrire transformé par la fonction en "unsigned char"

Remarque : Ces fonctions sont des macro définitions correspondantes à *fgetc(stdin)* et *fputc(carac, stdout)*.

La fonction ***ungetc()*** replace un caractère dans un tampon, ce qui rend ce caractère de nouveau disponible pour une relecture ultèrieure.

```
#include <stdio.h>                                  // I/O bufferisées
int ungetc (int carac, FILE *pt);
```

Explication des paramètres :

- int carac : caractère à replacer dans le tampon transformé par la fonction en "unsigned char"
- FILE *pt : pointeur du fichier renvoyé par la fonction ***fopen()***

Les fonctions ***fgets()*** et ***fputs()*** ont pour but d'aller respectivement lire et écrire une chaîne de caractères dans un fichier préalablement ouvert avec la fonction ***fopen()***.

```
#include <stdio.h>                                  // I/O bufferisées
char *fgets (char *ch, int taille, FILE *pt);
int fputs (char *ch, FILE *pt);
```

Explication des paramètres :

- char *ch : zone permettant de stocker la chaîne lue ou contenant la chaîne à écrire. La fonction "***fgets()***" rajoutera elle-même le caractère '\0' en fin de chaîne stockée et la fonction "***fputs()***" s'attend à recevoir une chaîne donc un tableau de caractères ayant une des valeurs à '\0'.
- int taille : taille maximale de la zone en octets. La lecture s'arrêtera à "taille – 1" (pour pgarder une place pour le caractère '\0') ou si elle rencontre un caractère '\n'.
- FILE *pt : pointeur du fichier renvoyé par la fonction ***fopen()***

Valeur renvoyée par fputs (int) :

- Nombre de caractères écrits
- Macro définition "EOF"si une erreur s'est produite

Valeur renvoyée par fgets (char*) :

- Pointeur vers la zone de stockage (correspond à la variable "chaîne")
- Constante "NULL" si une erreur s'est produite

Les fonctions ***gets()*** et ***puts()*** ont pour but d'aller respectivement lire et écrire une chaîne de caractères dans dans le tampon clavier ou écran.

```
#include <stdio.h>                                  // I/O bufferisées
char *gets (char *ch);
int fputs (char *ch);
```

Explication des paramètres :

- char *ch : zone permettant de stocker la chaîne lue ou contenant la chaîne à écrire. La fonction "***gets()***" rajoutera elle-même un caractère '\n' (fin de ligne) et le caractère '\0' en fin de chaîne stockée; et la fonction "***puts()***" s'attend à recevoir une chaîne donc un tableau de caractères ayant une des valeurs à '\0'.

Valeur renvoyée par puts (int) :

- Nombre de caractères écrits
- Macro définition "EOF"si une erreur s'est produite

Valeur renvoyée par gets (char*) :

- Pointeur vers la zone de stockage (correspond à la variable "chaîne")
- Constante "NULL" si une erreur s'est produite

Remarque : La fonction "***gets()***" est très dangereuse à utiliser du fait qu'elle ne limite pas la taille de la chaîne saisie… alors que la zone de stockage a une taille forcément limitée.

Les fonctions ***fread()*** et ***fwrite()*** ont pour but d'aller respectivement lire et écrire des octets dans un fichier préalablement ouvert avec la fonction ***fopen()***.

```
#include <sys/types.h>                                  // Création de nouveaux types
#include <stdio.h>                                      // I/O bufferisées
size_t fread (void *buffer, size_t taille, size_t nb, FILE *pt);
size_t fwrite (void *buffer, size_t taille, size_t nb, FILE *pt);
```

Explication des paramètres :

- void *buffer : pointeur vers une zone mémoire dans laquelle seront rangés (ou pris) les caractères lus (ou écrits) du fichier
- size_t taille : taille en octets d'un élément à lire ou écrire
- size_t nb : nombre d'éléments à lire ou écrire
- FILE *pt : pointeur du fichier renvoyé par la fonction ***fopen()***

Valeur renvoyée (size_t défini dans "sys/types.h>") :

- Le nombre d'éléments réellement lus (ou écrits) dans le fichier s'il y en a. Ce nombre peut être inférieur à ***nb*** (par exemple si on demande de lire plus que ce qu'il n'y a), mais jamais supérieur.
- 0 s'il n'y a plus rien à lire ou si une erreur s'est produite

Les fonctions ***fscanf()*** et ***fprintf()*** ont pour but d'aller respectivement saisir des variables à partir d'un fichier et écrire des chaînes formatées dans un fichier préalablement ouvert avec la fonction ***fopen()***.

```
#include <stdio.h>                                      // I/O bufferisées
int fscanf (FILE *pt, …);
int fprintf (FILE *pt, …);
```

Explication des paramètres :

- FILE *pt : pointeur du fichier renvoyé par la fonction ***fopen()***
- … format de saisie ou d'affichage correspondant aux fonctions "***scanf()***" et "***printf()***"

Valeur renvoyée (int) :

- Le nombre d'octets lus ou écrits dans le fichier (cela peut être "0" si rien n'a été fait)
- -1 si une erreur s'est produite

Les fonctions ***getline()*** et ***getdelim()*** ont pour but d'aller récupérer des lignes de taille indéfinies à partir d'un fichier préalablement ouvert avec la fonction ***fopen()***, une ligne étant une séquence de caractères terminée par le caractère '\n'.
La fonction ***getdelim()*** permet de spécifier un caractère particulier supplémentaire pour délimiter la fin de ligne.
Remarque: La taille de la ligne à récupérer étant inconnue à l'avance, le programmeur ne peut pas prévoir la mémoire pour stocker la ligne récupérée. Ce sont donc les fonctions elles-mêmes qui allouent (***malloc()*** et/ou ***realloc()***) suffisemment de mémoire pour stocker la ligne récupérée. Charge au programmeur qui utilise ces fonctions de libérer la zone de stockage (***free()***) une fois qu'elle est devenue inutile.

```
#include <sys/types.h>                                          // Création de nouveaux types
#include <stdio.h>                                              // I/O bufferisées
size_t getline (char **buffer, size_t *taille, FILE *pt);
size_t getdelim (char **buffer, size_t *taille, int delim, FILE *pt);
```

Explication des paramètres :

- char **buffer : pointeur sur un pointeur vers une zone mémoire dans laquelle seront rangés les caractères lus du fichier. La double indirection se justifie ici car la fonction doit pouvoir modifier l'adresse (pointeur) de la zone allouée. Si (*buffer) est nul, la fonction allouera la zone mémoire. Si (*buffer) n'est pas nul, il doit correspondre à une zone déjà allouée et la fonction aura possibilité d'agrandir cette zone avec une réallocation si nécessaire.
- size_t *taille : pointeur sur une variable contenant la taille de la zone allouée. Si la fonction modifie la taille de cette zone, elle stockera dans la variable pointée la nouvelle taille.
- int delim : Caractère permettant de délimiter une ligne
- FILE *pt : pointeur du fichier renvoyé par la fonction ***fopen()***

Valeur renvoyée (size_t défini dans "sys/types.h>") :

- Le nombre d'octets réellement lus du fichier s'il y en a.
- 0 s'il n'y a plus rien à lire ou si une erreur s'est produite

c) Autres fonctions

La fonction ***fflush()*** permet de finaliser les écritures encore en attentes dans les tampons mémoires associés aux sorties (fichiers en cours d'écriture, écran, etc.). Les tampons d'écritures sont alors physiquement recopiés sur les fichiers associés.

```
#include <stdio.h>                                              // I/O bufferisées
int fflush (FILE *pt);
```

Explication des paramètres :

- FILE *pt : pointeur du fichier renvoyé par la fonction ***fopen()***. Il est possible d'utiliser la macro définition "***NULL***" pour demander à la fonction de vider **tous** les tampons associés à des fichiers ouverts. Il est aussi possible de finaliser les écritures en attentes sur *stdout* (tampon associé aux messages normaux venant du programme) ou *stderr* (tampon associé aux messages d'erreur).

Valeur renvoyée (int) :

- 0 si elle réussi intégralement
- Macro définition "EOF"si une erreur s'est produite

Remarque : La fonction ***fflush()*** n'a été créée que pour les écritures en attentes et ne doit pas être appelée avec le buffer clavier *stdin* car son comportement devient alors indéterminé.
Cependant, certains systèmes (*Sun* par exemple) acceptent cet appel qui a alors pour effet de vider le clavier des données qu'il contient.

Les fonctions ***fpurge()*** et ***__fpurge()*** permettent de vider sans les traiter les tampons mémoires associés aux entrées-sorties. A la différence de "***fflush()***" qui finalise les données en attente d'écriture avant de les effacer, les fonctions ***fpurge()*** et ***__fpurge()*** suppriment simplement les données encore présentes dans les tampons associés aux entrées-sorties.
La fonction ***__fpurge()*** diffère de la fonction ***fpurge()*** en ne renvoyant rien.

```
#include <stdio.h>                                  // I/O bufferisées
int fpurge (FILE *pt);
void __fpurge(FILE *pt);
```

Explication des paramètres :

- FILE *pt : pointeur du fichier renvoyé par la fonction ***fopen()***. Il est possible d'utiliser la macro définition "***NULL***" pour demander à la fonction de purger **tous** les tampons associés à des fichiers ouverts. Il est aussi possible de purger les écritures en attentes sur *stdout* (tampon associé aux messages normaux venant du programme), *stderr* (tampon associé aux messages d'erreur) ou *stdin* (tampon associé aux données entrantes).

Valeur renvoyée (int pour *fpurge()*) :

- 0 si elle réussi intégralement
- Macro définition "EOF"si une erreur s'est produite

Remarque : Les fonctions ***fpurge()*** et ***__fpurge()*** n'ont été écrites que très récemment et ne se trouvent donc pas sur tous les systèmes. De plus, la nécessité de faire appel à ces fonctions est souvent dûe à une mauvaise conception du programme. Néanmoins, elles peuvent être remplacées, si le besoin s'en fait sentir, par la macro définition suivante :

```
#define FPURGE(flux)        ((flux)->_IO_read_ptr=NULL,\
                             (flux)->_IO_read_end=NULL,\
                             (flux)->_IO_read_base=NULL,\
                             (flux)->_IO_write_ptr=NULL,\
                             (flux)->_IO_write_end=NULL,\
                             (flux)->_IO_write_base=NULL)
…suite de l'entête
int main(void)
{
        … suite du code
        FPURGE(stdin);
        FPURGE(stdout);
        FPURGE(stderr);
        … suite du code
```

Les fonctions ***fseek()*** et ***ftell()*** permettent respectivement de changer et de renvoyer la position du pointeur interne de lecture et d'écriture d'un fichier préalablement ouvert avec la fonction ***fopen()***.

```
#include <stdio.h>                                  // I/O bufferisées
int fseek (FILE *pt, long offset, int whence);
int ftell (FILE *pt);
```

Explication des paramètres :

- FILE *pt : pointeur du fichier renvoyé par la fonction ***fopen()***
- long offset : position de placement dans le fichier
- int whence : origine devant être utilisée pour le calcul du positionnement. Cette origine doit être prise parmi l'une des trois constantes suivantes :
  - SEEK_SET : la position demandée sera calculée à partir du début du fichier
  - SEEK_CUR : la position demandée sera calculée à partir de la position courante
  - SEEK_END : la position demandée sera calculée à partir de la fin du fichier

Valeur renvoyée (int) :

- La nouvelle position (pour ***fseek()***) ou la position actuelle (pour ***ftell()***) mesurée en octets depuis le début du fichier.
- -1 si une erreur s'est produite

Les fonctions ***feof()***, ***ferror()*** et ***clearerr()*** permettent respectivement de vérifier l'indicateur de fin de fichier, l'indicateur d'erreur sur un fichier et réinitialiser ces deux indicateurs pour un fichier préalablement ouvert avec la fonction ***fopen()***.

```
#include <stdio.h>                                  // I/O bufferisées
int feof (FILE *pt);
int ferror (FILE *pt);
void clearerr (FILE *pt);
```

Explication des paramètres :

- FILE *pt : pointeur du fichier renvoyé par la fonction ***fopen()***

Valeur renvoyée (int) : Ces fonctions ne peuvent pas échouer et renvoient une valeur non-nulle si l'indicateur vérifié est actif.

La fonction ***fileno()*** renvoie un descripteur en mode "bas niveau" (*cf. cours sur la programmation en C système sous Unix*) d'un fichier préalablement ouvert avec la fonction ***fopen()***.

```
#include <stdio.h>                                  // I/O bufferisées
int fileno (FILE *pt);
```

Explication des paramètres :

- FILE *pt : pointeur du fichier renvoyé par la fonction ***fopen()***

Valeur renvoyée (int) : Cette fonction ne peut pas échouer et renvoie un descripteur utilisable par les fonctions d'accès "bas niveau" sur le fichier.

Les fonctions ***truncate()*** et ***ftruncate()*** permettent de couper un fichier à une taille donnée. La fonction ***truncate()*** travaille sur un fichier non ouvert alors que la fonction ***ftruncate()*** travaille sur un fichier ouvert en mode "bas niveau" (*cf. cours sur la programmation en C système sous Unix*).

```
#include <sys/types.h>                  // Création de nouveaux types
#include <stdlib.h>                     // Utilisation des librairies standard
int truncate (char *nom, off_t size);
int ftruncate (int fd, off_t size);
```

Explication des paramètres :

- char *nom : chaîne de caractères contenant le nom du fichier à couper
- int fd : Descripteur du fichier ouvert en mode "bas niveau"
- off_t size : Taille en octets à laquelle on veut couper le fichier

Valeur renvoyée (int) :

- La nouvelle taille en octets du fichier.
- -1 si une erreur s'est produite

d) Exercice

Faire un programme qui copie un fichier sous un autre nom (clone de la commande Unix "cp").
Les noms des fichiers seront passés au programme sous forme d'argument
Le programme ne vérifiera pas l'écrasement éventuel

```
#include <sys/types.h>                  // Création de nouveaux types
#include <stdio.h>                      // I/O bufferisées

#define SZ_BUFFER       (1024)          // Taille zone de stockage (<= 32768)

// Déclaration des fonctions utiles
void usage(char*);                      // Décrit comment utiliser le programme
size_t copie(char*, char*);             // Effectue la copie

// Fonction principale du programme
int main(
        int argc,                       // Nombre d'arguments passés au programme
        char *argv[])                   // Liste des arguments passés au programme
{
        // Déclaration des variables
        long NbOctet;                   // Nb Octets copiés

        // Vérification assez d'argument (il faut 2 noms pour pouvoir copier)
        if (argc <= 2)
        {
                // On indique que ce n'est pas bon
                printf("Pas assez d'argument pour %s\n", argv[0]);

                // On indique comment se servir du programme
                usage(argv[0]);

                // On quitte le programme avec une valeur différente de "0" (erreur)
                return 1;
        }

        // Ici, il y a assez d'argument – On appelle la fonction de copie et on vérifie
        if ((NbOctet=copie(argv[1], argv[2])) < 0)
        {
                // Ici, la copie s'est mal passée
                printf("Copie %s dans %s échouée\n", argv[1], argv[2]);
                return 2;
        }
```

```
	// La copie s'est bien passée - On affiche le nombre d'octets copiés
	printf("%s copié dans %s: %u octets\n", argv[1], argv[2], NbOctet);

	// Fin du programme
	return 0;
}

// Fonction de copie
size_t copie(
	char *NomSrc,				// Nom fichier source
	char *NomDest)				// Nom fichier destination
{
	// Déclaration des variables
	FILE *fpSrc;				// Ptr fichier source
	FILE *fpDest;				// Ptr fichier destination
	char buffer[SZ_BUFFER];			// Zone de stockage
	size_t nb_buffer;			// Nb octets traités en un bloc
	size_t nb_total;			// Nb octets écrits au total

	// Ouverture fichier source
	if ((fpSrc=fopen(NomSrc, "r")) == NULL)
	{
		// Le fichier n'a pas été ouvert
		printf("Erreur ouverture %s\n", NomSrc);

		// La fonction ne peut pas continuer
		return -1;
	}

	// Ouverture fichier destination
	if ((fpDest=fopen(NomDest, "w")) == NULL)
	{
		// Le fichier n'a pas été ouvert
		printf("Erreur ouverture %s\n", NomDest);

		// Fermeture fichier source (qui est toujours ouvert)
		fclose(fpSrc);

		// La fonction ne peut pas continuer
		return -2;
	}

	// Boucle de copie bloc par bloc
	nb_total=0;
	while ((nb_buffer=fread(buffer, 1, SZ_BUFFER, fpSrc)) != 0)
	{
		// On vérifie qu'il n'y a pas eu erreur de lecture
		if (nb_buffer < 0)
		{
			// La lecture a échouée
			printf("Erreur lecture après octet %ld\n", nb_total);

			// Fermeture fichiers (toujours ouverts)
			fclose(fpSrc);
			fclose(fpDest);

			// La fonction ne peut pas continuer
			return -3;
		}

		// On a lu un bloc de "nb_buffer" octets – On va écrire le même
```

```
                nb_total+=nb_buffer;

                // On vérifie qu'il n'y a pas eu erreur d'écriture
                if (fwrite(buffer, 1, nb_buffer, fpDest) < 0)
                {
                        // L'écriture a échouée
                        printf("Erreur écriture avant bloc %ld\n", nb_total);

                        // Fermeture fichiers (toujours ouverts)
                        fclose(fpSrc);
                        fclose(fpDest);

                        // La fonction ne peut pas continuer
                        return -4;
                }
        }

        // Tout s'est bien passé – Fermeture fichiers et fin de fonction
        fclose(fpSrc);
        fclose(fpDest);
        return nb_total;
}

// Fonction d'affichage de l'usage du programme
void usage(
        char *NomPGM)                                   // Nom programme à afficher
{
        // Déclaration des variables
        // Pas de variable

        // Affichage de l'usage à faire
        printf("usage: %s source dest\n", NomPGM);
}
```

Cet exercice montre bien la démarche importante du programmeur qui doit s'efforcer de toujours vérifier un maximum de cas possible (*cf. chapitre sur la gestion des exceptions*). De plus, sa modularité (séparation des différents grands traitements) lui permet d'être évolutif. Enfin utiliser une macro définition pour paramétrer la zone de stockage rend ce programme portable sur des ordinateurs plus ou moins puissants.

5) Gestion des exceptions

Une exception est une impossibilité au système d'effectuer une fonction demandée par l'utilisateur (ouvrir un fichier auquel on n'a pas accès; écrire alors que le fichier est ouvert en lecture; etc.).
D'une façon assez standard; chaque fois qu'une fonction de la librairie standard ne peut pas effectuer le travail pour lequel elle a été prévue; elle renvoie généralement :

- NULL si elle doit renvoyer un pointeur
- (-1) si elle doit renvoyer une valeur numérique

Cependant; il peut être intéressant pour le programmeur de connaître la raison exacte de l'échec en sachant qu'il peut y avoir plusieurs raisons d'échec pour une fonction. Or, la valeur renvoyée par la fonction indique juste l'échec mais n'indique pas la raison de l'échec.

a) La variable "extern int errno"

La variable "***errno***" de type *int* est une variable générale de travail du langage C. Elle a été créée dans et pour la librairie standard **"/usr/lib/libc.a**" et est; de ce fait; disponible pour tout programme ou pour toute fonction qui sera compilé avec cette dernière.
Ce nombre entier est rempli automatiquement par toute fonction standard qui subi un échec dans une tentative interne de travail. Elle rempli ainsi la variable avec une valeur numérique standardisée indiquant la cause de son échec.
Le programmeur, ayant détecté l'échec de sa fonction appelée par un test de la valeur renvoyée ; et ayant lui-aussi accès à cette variable ; peux afficher sa valeur. Enfin; il ne lui reste qu'à consulter la table des exceptions dans sa documentation système pour connaître la raison exacte de l'échec.
Pour avoir accès à cette variable; il doit soit :

- la déclarer en "*extern*" puisqu'elle est déjà définie (déjà existante)
- inclure le fichier "*/usr/include/errno.h*" qui est un fichier d'en-tête déclarant, entre autres, cette variable

```
extern int errno;                                   // Ou bien "#include <errno.h>"

main()
{
        … début du code…
        fp=fopen("fichier", "w");                   // On tente d'ouvrir le fichier pour y écrire
        if (fp == NULL)                             // Si la fonction a échouée (fichier non ouvert)
        {
                // Affichage d'un message quelconque incluant la variable "errno"
                printf("Erreur ouverture fichier – Raison = %d\n", errno);

                // Sortie du programme (en général il est inutile d'aller plus loin)
                exit(1);
        }
        … suite du code …
}
```

Remarque : La valeur de "errno" n'a de signification qu'après un appel à une fonction ayant échoué. Elle n'est jamais "remise à zéro" quand une fonction n'échoue pas (cela ne sert à rien) ; mais rien n'interdit de la mettre manuellement à "zéro" si un hypothétique besoin s'en faisait sentir.

b) La variable "extern const char* const sys_errlist[]"

Le soucis de l'utilisation immédiate de la variable "***errno***" est son coté impersonnel. En effet, il est fortement rébarbatif d'exploiter un code numérique pour connaître la raison de l'échec d'une fonction ; même en allant chercher le libellé correspondant dans une obscure documentation système par ailleurs peut-être indisponible ou d'accès difficile.
C'est pourquoi il a été associé à la variable numérique "***errno***" un tableau de chaînes de caractères "***sys_errlist***" contenant tous les messages des échecs possibles de toutes les fonctions des librairies associées au programme.
Bien entendu, ce tableau étant associé à la variable "*errno*", il lui est fortement lié. Ainsi; le message correspondant à l'exception "n" sera stocké dans l'indice "n" du tableau.
Cependant, la déclaration du tableau ne se trouve dans aucun des fichiers d'en-têtes du système Unix. C'est à dire que le programmeur qui veut s'en servir doit impérativement déclarer cette variable qui est; rappelons-le; un tableau invariant de chaînes de caractères invariantes de longueurs non-fixées; donc un tableau constant de pointeurs constants.
Cette omission a été réparée dans le système "*Linux*" où la variable "***sys_errlist***" a été déclarée dans le fichier d'en-tête "***/usr/include/stdio.h***".

```
extern int errno;                                   // Ou bien "#include <errno.h>"
extern const char* const sys_errlist[];             // Ou bien "#include <stdio.h>" sous "Linux"

int main(void)
{
        … début du code…
        fp=fopen("fichier", "w");                   // On tente d'ouvrir le fichier pour y écrire
        if (fp == NULL)                             // Si la fonction a échouée (fichier non ouvert)
        {
                // Affichage d'un message quelconque propre au programmeur
                printf("Erreur ouverture fichier\n");

                // Affichage du message système
                printf("%s\n", sys_errlist[errno]);

                // Sortie du programme (en général il est inutile d'aller plus loin)
                exit(1);
        }
        … suite du code …
}
```

c) La variable "extern int sys_nerr"

La variable numérique "***sys_nerr***" de type *int* contient le nombre de messages d'erreurs disponibles dans le tableau "***sys_errlist[]***". Elle correspond de ce fait à la valeur maximale que peut prendre la variable "***errno***".
L'utilité de cette variable est très faible. Elle permet simplement d'avoir une limite de boucle pour celui qui souhaite afficher tous les messages d'erreurs disponibles dans son système; ou bien d'avoir un contrôle supplémentaire en vérifiant que "***errno***" est bien dans les limites imposées avant d'aller chercher le message "***sys_errlist[errno]***".
Comme pour la variable "***sys_errlist***"; la variable "***sys_nerr***" n'est déclarée nulle part en dehors du fichier "***/usr/include/stdio.h***" du système "*Linux*".

d) La fonction "strerror"

La fonction ***strerror()*** renvoie l'adresse du message d'erreur correspondant à "**errno**". Cette fonction correspond donc exactement au tableau "**sys_errlist**" mais elle lui est préférable pour des raisons d'évolutivité et de maintenance.

```
#include <string.h>                                    // Gestion des chaînes de caractères
char *strerror(unsigned int errnum);
```

Explication des paramètres :

- unsigned int errnum : Le numéro de l'erreur dont on veut le libellé

Valeur renvoyée (int) :

- Un pointeur sur le libellé correspondant à l'erreur "*errnum*"

```
#include <errno.h>

int main(void)
{
        … début du code…
        fp=fopen("fichier", "w");                      // On tente d'ouvrir le fichier pour y écrire
        if (fp == NULL)                                // Si la fonction a échouée (fichier non ouvert)
        {
                // Affichage d'un message quelconque propre au programmeur
                printf("Erreur ouverture fichier\n");

                // Affichage du message système
                printf("%s\n", strerror(errno));

                // Sortie du programme (en général il est inutile d'aller plus loin)
                exit(1);
        }
        … suite du code …
}
```

e) Fonction "perror()"

La fonction ***perror()*** affiche sur la sortie des erreurs un texte choisi par le programmeur immédiatement suivi du message d'erreur correspondant à ***errno***. Ce n'est qu'un affichage habillé de ***sys_errlist[]***.

```
#include <errno.h>
void perror (const char *txt);
```

Explication des paramètres :

☞ const char *txt : Chaîne contenant un message qui sera affiché en plus du message provenant de ***sys_errlist[]***

Exemple :

```
#include <errno.h>

int main(void)
{
	… début du code…
	fp=fopen("fichier", "w");			// On tente d'ouvrir le fichier pour y écrire
	if (fp == NULL)					// Si la fonction a échouée (fichier non ouvert)
	{
		// Affichage d'un message quelconque suivi automatiquement du message système
		perror("Erreur ouverture fichier");

		// Sortie du programme (en général il est inutile d'aller plus loin)
		exit(1);
	}
	… suite du code …
}
```

6) Fonctions mathématiques

Pour utiliser ces fonctions, il est nécessaire, lors de la création de l'exécutable, de lancer le compilateur en lui demandant d'inclure la librairie mathématique "**/usr/lib/libm.a**".
Exemple :

```
cc prog.c /usr/lib/libm.a –o prog
cc prog.c –lm –o prog
```

Liste des fonctions mathématiques en vrac.

```
#include <math.h>                          // Mathématiques en virgule flotante
double exp (double x);                     // Exponentielle
double log (double x);                     // Logarithme base 2
double log10 (double x);                   // Logarithme base 10
double pow (double x, double y);           // "x" puissance "y"
double sqrt (double x);                    // Racine carrée
double sin (double x);                     // Sinus (x doit être en radians)
double cos (double x);                     // Cosinus (x doit être en radians)
double tan (double x);                     // Tangente (x doit être en radians)
double asin (double x);                    // Arc sinus (résultat en radians)
double acos (double x);                    // Arc cosinus (résultat en radians)
double atan (double x);                    // Arc tangente (résultat en radians)
```

Les fonctions décrites précédemment peuvent échouer dans certains cas (racine carrée d'un nombre négatif ou bien élévation d'un nombre négatif à une puissance non-entière (ce qui peut donner un nombre complexe), logarithme d'un nombre négatif, ou bien tangente de $\pm\pi/2$ qui n'existe pas, etc.).
A ce moment là ; les fonctions renvoient une valeur spécifique "*NAN*" (Not a Number) et positionnent en général ***errno*** avec l'erreur "*EDOM*".
La fonction ***isnan()*** permet de vérifier si la valeur renvoyée par une fonction mathématique correspond à "*NAN*".

```
#include <math.h>                          // Mathématiques en virgule flotante
int isnan (double x);                      // Vérifie si "x" est "not a number"
```

Exemple :

```
#include <math.h>                          // Mathématiques en virgule flotante
int main(void)
{
        double tangente=…;
        double angle;

        angle=atan(tangente);
        if (isnan(angle))
                printf("Erreur atan(%lf)\n", tangente);
        … suite du code
}
```

XV) ANNEXES

1) Priorités des opérateurs (par ordre décroissant)

Priorité	Opérateurs	Signification	Associativité
15	() [] -> .	Expression Indice Membre d'une structure pointée Membre d'une structure	de gauche à droite
14	! ~ ++ -- + - (type) * & sizeof()	Non logique Complément à un Incrémentation Décrémentation Signe (opérateur unaire) Casting Elément pointé Adresse de Taille de	de droite à gauche
13	* / %	Multiplication Division Modulo	de gauche à droite
12	+ -	Addition Soustraction	de gauche à droite
11	<< >>	Décalage à gauche Décalage à droite	de gauche à droite
10	< > <= >=	Inférieur Supérieur Inférieur ou égal à Supérieur ou égal à	de gauche à droite
9	== !=	Egal à Différent de	de gauche à droite
8	&	ET bit à bit	de gauche à droite
7	^	OU exclusif bit à bit	de gauche à droite
6	|	OU bit à bit	de gauche à droite
5	&&	ET logique	de gauche à droite
4	||	OU logique	de gauche à droite
3	? :	Expression conditionnelle (ternaire)	de droite à gauche
2	= += -= *= /= %= <<= >>=&= ^= |=	Affectation Affectations diverses	de droite à gauche
1	,	Opérateur séquentiel	de gauche à droite

2) Options de "printf" et "scanf"

%	Printf	scanf
%c	Affichage au format "char"	Saisie d'un "char"
%d	Affichage au format "int"	Saisie d'un "int"
%hd	Affichage au format "short"	Saisie d'un "short int"
%ld	Affichage au format "long"	Saisie d'un "long"
%lld	Affichage au format "long long"	Saisie d'un "long long"
%u	Affichage au format "unsigned int"	Saisie d'un "unsigned int"
%hu	Affichage au format "unsigned short"	Saisie d'un "unsigned short"
%lu	Affichage au format "unsigned long"	Saisie d'un "unsigned long"
%llu	Affichage au format "unsigned long long"	Saisie d'un "unsigned long long"
%o	Affichage au format "int" en octal	Saisie d'un "int" en octal
%ho	Affichage au format "short" en octal	Saisie d'un "short" en octal
%lo	Affichage au format "long" en octal	Saisie d'un "long" en octal
%llo	Affichage au format "long long" en octal	Saisie d'un "long long" en octal
%x	Affichage au format "int" en hexadécimal	Saisie d'un "int" en hexadécimal
%hx	Affichage au format "short" en hexadécimal	Saisie d'un "short" en hexadécimal
%lx	Affichage au format "long" en hexadécimal	Saisie d'un "long" en hexadécimal
%llx	Affichage au format "long long" en hexadécimal	Saisie d'un "long long" en hexadécimal
%f	Affichage au format "float" en notation "classique"	Saisie d'un "float" en notation "classique"
%lf	Affichage au format "double" en notation "classique"	Saisie d'un "double" en notation "classique"
%Lf	Affichage au format "long double" en notation "classique"	Saisie d'un "long double" en notation "classique"
%e	Affichage au format "float" en notation "scientifique"	Saisie d'un "float" en notation "scientifique"
%le	Affichage au format "double" en notation "scientifique"	Saisie d'un "double" en notation "scientifique"
%Le	Affichage au format "long double" en notation "scientifique"	Saisie d'un "long double" en notation "scientifique"
%g	Affichage au format "float" en notation classique ou scientifique ("printf" choisira l'affichage prenant le moins de place)	
%lg	Affichage au format "double" en notation classique ou scientifique ("printf" choisira l'affichage prenant le moins de place)	
%Lg	Affichage au format "long double" en notation classique ou scientifique ("printf" choisira l'affichage prenant le moins de place)	
%s	Affichage d'une chaîne de caractères	Saisie d'une chaîne de caractères
%r	Affichage au format "chiffres romains" minuscules (option non normalisée)	
%R	Affichage au format "chiffres romains majuscules" (option non normalisée)	

3) Les problèmes les plus fréquents

a) Mes affichages ne se font pas au bon moment

Voici un exemple de code qui est correct… mais qui présente le léger défaut de ne rien affichar avant qu'on ait saisi la valeur demandée.

```
#include <stdio.h>                                        // I/O bufferisées

main()
{
        int i;

        printf("Entrez une valeur : ");
        scanf(%d", &i).

        printf("La valeur que vous avez entré est %d\n", i);
}
```

Le problème vient de la fonction "printf". Cette fonction est "bufférisée"; c'est à dire qu'elle stocke les données à afficher dans une mémoire (buffer) afin de réduire les requêtes d'entrées-sorties qui sont (relativement) très longues. Les données ne sont alors réellement affichées (sorties) que dans les cas suivants :

- La fonction rencontre un caractère '\n' dans les données à afficher
- Le buffer est plein
- Le programme se termine
- Le buffer est vidé sur demande explicite du programmeur

Si le programmeur désire maîtriser dans son déroulement du programme le moment où seront affichées ses données; il doit provoquer un des cas ci-dessus. Le plus facile est donc de demander le vidage du buffer en employant la fonction "*fflush()*" et en lui indiquant qu'il faut vider le buffer de sortie standard "*stdout*".

Voici le code rectifié…

```
#include <stdio.h>                                        // I/O bufferisées

main()
{
        int i;

        printf("Entrez une valeur : "); fflush(stdout);
        scanf("%d", &i).

        printf("La valeur que vous avez entré est %d\n", i);
}
```

Le problème se présente aussi parfois dans l'écriture d'un fichier avec l'emploi de la fonction "*fprintf()*". Le fichier ne sera réellement écrit sur disque que dans les cas ci-dessus ou bien lors de sa fermeture par la fonction "*fclose()*".

b) Mes saisies se font une fois sur deux

Voici un exemple de code qui a l'air correct… mais qui présente le défaut de ne pas laisser à l'opérateur saisir sa demande de quitter la boucle.

```
#include <stdio.h>                                        // I/O bufferisées

int main(void)
{
        int age;
        char c;

        do {
                printf("Age :"); fflush(stdout);
                scanf("%d", &age);

                printf("Fin (o/n) :"); fflush(stdout);
                scanf("%c", &c);

                printf("age=<%d>, fin=<%c>\n", age, c);
        } while (c != 'o');
}
```

Le problème vient de la saisie du numérique. Lorsque l'opérateur entre son nombre, il valide sa saisie par l'appui de la touche "Entrée". Or, cette touche envoie elle-aussi un caractère dans le tampon associé au clavier.
Lorsque la fonction "*scanf()*" extrait le nombre du tampon associé au clavier, elle s'arrête au premier caractère qui n'est plus numérique donc s'arrête à ce fameux caractère "Entrée" qui reste dans le tampon.
Lors de la saisie suivante, la fonction "*scanf()*" ayant déjà un caractère à traiter ne demande donc rien à l'opérateur.
Pour que le buffer arrive vide à l'entrée de "*scanf()*", il convient, après la saisie du numérique, d'extraire ce caractère inutile par l'appel d'une fonction appropriée ("*getchar()*" par exemple).
L'appel à la fonction "*fpurge()*" est envisageable mais bien inutile du fait de la cause de cette erreur très simple à rectifier.

Voici le code rectifié…

```
#include <stdio.h>                                        // I/O bufferisées

int main(void)
{
        int age;
        char c;

        do {
                printf("Age :"); fflush(stdout);
                scanf("%d", &age);
                getchar();

                printf("Fin (o/n) :"); fflush(stdout);
                scanf("%c", &c);

                printf("age=<%d>, fin=<%c>\n", age, c);
        } while (fin != 'o');
}
```

c) Mes boucles ne s'arrêtent pas

Dans l'emploi des boucles, il faut se méfier de la zone des valeurs possibles des variables qu'on utilise car une variable qui atteint sa valeur limite maximale et qui est incrémentée de "1" repasse à sa valeur minimale sans que le programme ou le compilateur indique une erreur quelconque.
Il en va de même pour une variable qui atteint sa valeur limite minimale et qui est décrémentée de "1". Elle repasse alors à sa valeur limite maximale.
Voici différents exemples de boucles infinies…

```
char i;
for (i=0; i < 200; i++)
        printf("i=%d\n", i);
// Boucle infinie car un "char" (signed) varie de –128 à 127 et sera toujours inférieur à 200
```

```
unsigned short i;
for (i=10; i >= 0; i--)
        printf("i=%d\n", i);
// Boucle infinie car un "unsigned" est toujours positif ou nul
```

```
long i=10;
do {
        printf("i=%d\n", i);
        i--;
} while (i > 1 >= 0);
//      Cette boucle réellement bizarre montre bien qu'en C, on peut écrire vraiment
        n'importe-quoi pourvu que ce soit syntaxiquement correct.
        La condition de fin sera évaluée de gauche à droite. On aura donc :
        i > 1 (vrai ou faux) => renvoie un nombre "x" différent de 0 ou x=0
        x >= 0 sera toujours vrai => Boucle infinie                                    */
```

d) Les fonctions de la librairie mathématique renvoient des valeurs incohérentes

N'oubliez pas d'inclure les déclarations des fonctions mathématiques "#include <math.h>" au début de chaque fichier source devant utiliser une fonction de la librairie mathématique.
En effet, lorsque le compilateur arrive sur une fonction non connue (non déclarée), il la déclare automatiquement par défaut en fonction de type "int" (norme *ANSI*). Ensuite, lors de l'appel au code exécutif de la fonction situé dans la librairie mathématique, le résultat du calcul est transformé dans le type de la fonction (int) et il perd toute sa cohérence.
L'inclusion de "math.h" permet de déclarer au compilateur toutes les fonctions qui seront utilisées.

INDEX