Cours : langage C

1. Présentation du langage C

1.1. Historique

Langage de programmation développé en 1970 par Dennie Ritchie aux Laboratoires **Bell** d'**AT&T.** Il est l'aboutissement de deux langages :

- **BPCL** développé en 1967 par Martin Richards.
- **B** développé en 1970 chez **AT&T** par Ken Thompson.

Il fut limité à l'usage interne de **Bell** jusqu'en 1978 date à laquelle Brian Kernighan et Dennie Ritchie publièrent les spécifications définitives du langage : «~**The C programming Language**~».

Au milieu des années 1980 la popularité du langage était établie. De nombreux compilateurs ont été écrits, mais comportant quelques incompatibilités portant atteinte à l'objectif de portabilité. Il s'est ensuivi un travail de normalisation effectué par l'**ANSI** (American National Standard Institue) qui a abouti en 1988 avec la parution par la suite du manuel : «~**The C programming Language - 2ème édition**~».

1.2. Intérêts du langage

- Langage polyvalent permettant le développement de systèmes d'exploitation, de programmes applicatifs scientifiques et de gestion.
- Langage structuré.
- Langage évolué qui permet néanmoins d'effectuer des opérations de bas niveau («~assembleur d'Unix~»).
- Portabilité (en respectant la norme !) due à l'emploi de bibliothèques dans lesquelles sont reléguées les fonctionnalités liées à la machine.
- Grande efficacité et puissance.

1.3. Qualités attendues d'un programme

- Clarté
- Simplicité
- Modularité
- Extensibilité

2. Généralités

2.1. Jeu de caractères

Le jeu de caractères utilisé pour ecrire une programme est composé de :

- 26 lettres de l'alphabet (minuscules, majuscules)
- chiffres 0 à 9
- séquences d'échappement telles :
  - passage à la ligne (**\n**),
  - tabulation (**\t**),
  - backspace (**\b**).

- caractères spéciaux :

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

2.2. Structure d'un programme C

- Un Programme C :
  est un fichier source entrant dans la composition d'un programme exécutable est fait d'une succession d'un nombre quelconque d'éléments indépendants, qui sont :
  - des directives pour le préprocesseur (lignes commençant par #),
  - des constructions de types (***struct, union, enum, typedef***),
  - des déclarations de variables et de fonctions externes,
  - des définitions de variables et
  - des définitions de fonctions dont l'une doit s'appeler ***main*** et apparaître obligatoirement dans le programme.
- Une Fonction est formé:
  - d'entête (type et nom de la **fonction** suivis d'une liste d'arguments entre parenthèses),
  - instruction composée constituant le corps de la **fonction**.
- Instruction composée : délimitée par les caractères **{** et **}**
- Instruction simple : se termine par **;**
- commentaire : encadré par les délimiteurs **/*** et ***/**
- Instruction préprocesseur : commence par **#**

Exemple :

```
#include<stdio.h>
#define  PI 3.14159
/*  calcul de la surface d'un cercle */
main()
{
  float rayon, surface;
  float calcul(float rayon);

  printf("Rayon = ? ");
  scanf("%f", &rayon);
  surface = calcul(rayon);
  printf("Surface = %f\n", surface);
}
/* définition de fonction */
float calcul(float r)
{
/* définition de la variable locale */
  float a;

  a = PI * r * r;
  return(a);
}
```

2.3. Compilation et édition des liens

La transformation d'un texte écrit en langage C en un programme exécutable par l'ordinateur se fait en deux étapes ***: la compilation et l'édition de liens***.

La compilation est la traduction des fonctions écrites en C en des procédures équivalentes écrites dans un langage dont la machine peut exécuter les instructions. Le compilateur lit toujours un fichier, appelé ***fichier source***, et produit un fichier, dit ***fichier objet.***

Chaque fichier objet est incomplet, insuffisant pour être exécuté, car il contient des appels de fonctions ou des références à des variables qui ne sont pas définies dans le même fichier. Par exemple, le premier programme que vous écrirez contiendra déjà la fonction printf que vous n'aurez certainement pas écrite vous-même. L'édition de liens est l'opération par laquelle plusieurs fichiers objets sont mis ensemble pour se compléter mutuellement : un fichier apporte des définitions de fonctions et de variables auxquelles un autre fichier fait référence et réciproquement. L'éditeur de liens (ou linker) prend en entrée plusieurs fichiers objets et bibliothèques (une variété particulière de fichiers objets) et produit un unique ***fichier exécutable***.

2.4. Identificateurs et mots-clés

- Identificateur :
  nom donné aux diverses composantes d'un programme ; *variables, constante, tableaux, fonctions*.
  - Formé de lettres et de chiffres ainsi que du caractère **_** permettant une plus grande lisibilité. Le 1er caractère doit obligatoirement être une lettre ou bien le caractère **_**.
  - Peut contenir jusqu'à 31 caractères minuscules et majuscules.
  - Les minuscules et les majuscules ne sont pas equivalentes : les identificateurs *ligne* et *Ligne* sont différents.

Exemples

- Identificateurs valides :

x	y12	somme_1	_temperature
noms	surface	fin_de_fichier	TABLE

- Identificateurs invalides :

4eme	commence par un chiffre
x#y	caractère non autorisé (#)
no-commande	caractère non autorisé (-)
taux change	caractère non autorisé (espace)

- Mots réservés (mots-clés)

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			

2.5. 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

2.6. 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 :

Les compilateurs récents acceptent, comme commentaire, la séquence "//" spécifique au langage "C++" 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 la séquence "//"
```

2.7. 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.

3. Élements de base pour debuter

3.1. Les types de base (types simples ou scalaires)

Un programme manipule des données de différents types sous la forme de constantes ou de variables. 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. Les principaux types de base sont:

3.1.1. 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 $2^{63} - 1$; 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). Une variable est signée par défaut donc le mot clef *"**signed**"* est inutile.

3.1.2. 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.

3.1.3. 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".

Exemple

```
#include <stdio.h>
unsigned char mask;
long          val;
main()
{
  unsigned int indice;
  float        x;
  double       y;
  char         c;
        ...
  return;
}


double f(double x)
{
  unsigned short taille;
  int            i;
  unsigned long  temps;
  double         y;
       ...
  return y;
}
```

3.2. les fonctions d'entrée-sortie (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 "scan( )" 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);
}
```

3.3. 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 "O***x***" ou "0***X***". Ex : ox5 (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 12.43 -0.38.38 47. .27
  - notation scientifique : un nombre avec un "***e***" ou un "E" indiquant l'exposant du facteur "10".
    Ex : 3e18 (3 x 10^{18}).

```
4.25E4      4.25e+4      42.5E3
54.27E-32   542.7E-33    5427e-34
.48e13      48.e13       48.0E13
```

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

3.4. Les énumérations de constantes

Les **énumérations** sont des types définissant un ensemble de constantes qui portent un nom que l'on appelle **énumérateur**. Elles servent à rajouter du sens à de simples numéros, à définir des variables qui ne peuvent prendre leur valeur que dans un ensemble fini de valeurs possibles identifiées par un nom symbolique.

Syntaxe

```
enum [nom]
{
   énumérateur1,
   énumérateur2,
   énumérateur3,
        ...
   énumérateurn
};
```

Les constantes figurant dans les **énumérations** ont une valeur entière affectée de façon automatique par le compilateur en partant de **0** par défaut et avec une progression de **1**. Les valeurs initiales peuvent être forcées lors de la définition.

```
enum couleurs {noir, bleu, vert, rouge, blanc,jaune};
enum couleurs
{
   noir = -1,
   bleu,
   vert,
   rouge = 5,
```

```
   blanc,
   jaune
};
```

Dans le 1er exemple, les valeurs générées par le compilateur seront :

noir	**0**	vert	**2**	blanc	**4**
bleu	**1**	rouge	**3**	jaune	**5**

et dans le 2e :

noir	**-1**	vert	**1**	blanc	**6**
bleu	**0**	rouge	**5**	jaune	**7**

3.5. 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

3.5.1. 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 , k = j = 6
- *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 nième bit de la première valeur étant comparé au nième 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 nième bit de la première valeur étant comparé au nième 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 nième bit de la première valeur étant comparé au nième 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 23.

- *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 24.

3.5.2. 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 "=").

3.5.3. 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").

3.5.4. 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").

3.5.5. 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").

3.5.6. 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").

3.5.7. Exercices

Calculez les résultats des expressions suivantes :

```
(2 + 3) * 4                              /* Réponse : 20 */

(2 << 4) + 3                             /* Réponse : 35 */

1 | 3                                    /* Réponse : 3 */

(2 << 1) > 3 ?25 >> 2 : 25 >> 1          /* Réponse : 6 */

4 > 5                                    /* Réponse : 0 ("faux") */

2,3,4+5                                  /* Réponse : 9 (2 et 3 ne sont pas traités) */
```

4. LES INSTRUCTIONS DE CONTROLE

Dans un programme, les instructions sont exécutées séquentiellement, *c'est-à-dire dans l'ordre où elles apparaissent*
Or la puissance et le "comportement intelligent" d'un programme proviennent essentiellement :

- de la possibilité d'effectuer des "**choix**", de se comporter différemment suivant les "circonstances" (celles-ci pouvant être, par exemple, une réponse de l'utilisateur, un résultat de calcul...),
- de la possibilité d'effectuer des "**boucles**", autrement dit de répéter plusieurs fois un ensemble donné d'instructions.

Ces possibilités sont offertes par des instructions, nommées "**instructions de contrôle**", permettant de réaliser ces choix ou ces boucles.

- des **choix** : instructions ***if...else et switch***,
- des **boucles** : instructions ***do...while, while et for***.

4.1. l'instruction if...else

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

4.1.1. Syntaxe de l'instruction if

Le mot else et l'instruction qu'il introduit sont facultatifs, de sorte que cette instruction présente deux formes.

if (*expression***)**
  instruction_1
else
  instruction_2
	if (*expression***)**
	  instruction_1

Avec :

expression : expression quelconque

instruction__1 et instruction_2 : instructions quelconques, c'est-à-dire :

- simple (terminée par un point virgule),
- bloc,
- instruction structurée.

4.1.2. exemples

Partie de programme exécuté	***Résultat après exécution***

`if (5<3)`
`  printf("Vrai");`
`else`
`  printf("Faux");`
	`Faux`

`if (5!=3)`
`  printf("Vrai");`
`else`
`  printf("Faux");`
	`Vrai`

Partie de programme exécuté	***Résultat après exécution***

`if (5==3)`
`  printf("Vrai");`
`else`
`  printf("Faux");`
	`Faux`

`if (5>=5)`
`  printf("Vrai");`
`else`
`  printf("Faux");`
	`Vrai`

L'expression conditionnant le choix est quelconque. La richesse de la notion d'express; en C fait que celle-ci peut elle-même réaliser certaines actions. Ainsi :

```
if ( ++i < limite)    printf ("OK") ;
```

est équivalent à :

```
i = i + 1 ;
if ( i < limite )    printf ("OK") ;
```

Par ailleurs :

```
if ( i++ < limite )
```

est équivalent à :

```
i = i + 1 ;
if (  i-1 < limite )
```

De même :

```
if { ( c=getchar() ) != '\n' )
```

peut remplacer :

```
c = getchar() ;
if ( c != '\n' )
```

4.1.3. Exercices d'application:

1. Ecrire le programme qui permet de lire deux nombres réel x et y et de les afficher dans l'ordre croissant.
2. Ecrire un programme qui fait la même chose qu'au précédent pour trois nombres x, y et z.
3. Ecrire un programme qui lie un caractère et vérifie s'il s'agit du caractère de tabulation ou retour à la ligne.
4. Ecrire un programme qui permet de lire un nombre et indique s'il est positif, nul ou négatif.

4.1.4. Cas d'imbrication des instructions if

Nous avons déjà mentionné que les instructions figurant dans chaque partie du choix d'une instruction pouvaient être absolument quelconques. En particulier, elles peuvent, à leur tour, renfermer d'autres instructions ***if***. Or, compte tenu de ce que cette instruction peut comporter ou ne pas comporter de ***else***, il existe certaines situations où une ambiguïté apparaît C'est le cas dans cet exemple :

```
if (a<=b)    if (b<=c)   printf ("ordonné") ;
else printf ("non ordonné") ;
```

est-il interprété comme le suggère cette présentation ?

```
if (a<=b)
             if (b<=c) printf ("ordonné") ;
else   printf ("non ordonné") ;
```

ou bien comme le suggère celle-ci ?

```
if (a<=b)
             if (b<=c) printf ("ordonné") ;
             else printf ("non ordonné") ;
```

La première interprétation conduirait à afficher "non ordonné" lorsque la condition a<=b est fausse, tandis que la seconde n'afficherait rien dans ce cas.

La règle adoptée par le langage C pour lever une telle ambiguïté est la suivante :

Un ***else*** se rapporte toujours au dernier ***if*** rencontré auquel un ***else*** n'a pas encore été attribué.

Dans notre exemple, c'est la seconde présentation qui suggère le mieux ce qui se passe.

4.1.5. Exemple de cas d'imbrication des instructions if:

soit à écrire un programme de facturation avec remise, qui doit lire le prix hors taxes et calcule le prix TTC correspondant (avec un taux de TVA constant de 18,6%), et qui établit ensuite une remise dont le taux dépend de la valeur ainsi obtenue, à savoir :

- 0 % pour un montant inférieur à 1 000 DH
- 1 % pour un montant supérieur ou égal à 1 000 DH et inférieur à 2 000 DH
- 3 % pour un montant supérieur ou égal à 2 000 DH et inférieur à 5 000 DH
- 5 % pour un montant supérieur ou égal à 5 000 DH Ce programme est accompagné de deux exemples d'exécution.

```
#include<stdio.h>
#define TAUX_TVA    18.6
Main()
{
double ht, ttc, net, tauxr, remise ;
       printf("donnez le prix hors taxes : ") ;
       scanf ("%lf", &ht) ;
       ttc = ht * ( 1. + TAUX_TVA/100.) ;
       if ( ttc < 1000.)               tauxr  = 0 ;
       else   if ( ttc < 2000 )        tauxr  = 1. ;
              else   if ( ttc < 5000 )  tauxr = 3. ;
                     else               tauxr  =5. ;
       remise = ttc * tauxr /100. ;
       net = ttc - remise ;
       printf ("prix ttc         = %10.2lf\n", ttc) ;
       printf ("remise           = %10.2lf\n", remise) ;
       printf ("net à payer      = %10.2lf\n", net) ;
}
```

Exemple Nº1 d'exécution :

```
       donnez le prix hors taxes : 500
       prix ttc      593.00
       remise         0.00
       net à payer    593.00
```

Exemple Nº2 d'exécution :

```
       donnez le prix hors taxes : 4000
       prix ttc     4744.00
       remise        142.32
       net è payer   4601.68
```

4.2. L'INSTRUCTION SWITCH

L'instruction switch regarde si la valeur d'une expression fait partie d'un certain ensemble de constantes entières et effectue les traitements associés à la valeur correspondante.

4.2.1. Syntaxe:

```
switch(expression)
{
case expression-constante_1: [ suite_d'instructions_1 ];
case expression-constante_2: [ suite_d'instructions_2 ];
...
case expression-constante_n: [ suite_d'instructions_n ];
[ default: suite_d'instructions; ]
}
```

Avec:

`expression` : expression entière quelconque,

`expression-constante` *:* expression constante d'un type entier quelconque *(char* est accepté car il sera converti en *int),*

`suite_d'instructions` : séquence d'instructions quelconques.

N.B. : les crochets ([et]) signifient que ce qu'ils renferment est facultatif.

Chaque cas possible est étiqueté par une ou plusieurs constantes entières. L'expression est évaluée. Si elle correspond à une expression-constante, l'exécution commence par les instructions associées à cette étiquette.

L'instruction *break* permet de sortir immédiatement du switch. En effet lorsque le traitement d'un cas est terminé, l'exécution continue par le cas suivant. Pour différencier les divers cas il est donc nécessaire de terminer chacun d'entre eux par l'instruction break.

4.2.2. Exemples d'introduction de l'instruction switch

a) Premier exemple

Voyez ce premier exemple de programme accompagné de trois exemples d'exécution.

```
#include<stdio.h>
main()
{
   int n ;
   printf ("donnez un entier : ") ;
   scanf ("%d", &n) ;

   switch (n)
   {
      case 0 : printf ("nul\n") ;
      break ;
      case 1 : printf ("un\n") ;
      break ;
      case 2 : printf ("deux\n") ;
      break ;
   }
   printf ("au revoir\n");
}
```

Exemples d'exécution de ce programme

```
donnez un entier : 0
nul
au revoir

donnez un entier : 2
deux
au revoir

donnez un entier : 5
au revoir
```

b) deuxième exemple

Notez bien que le rôle de l'instruction *break* est fondamental. Voyez, à titre d'exemple, ce que produirait ce même programme en l'absence d'instructions break :

```
#include<stdio.h>
main()
{
   int n ;
   printf ("donnez un entier :  ");
   scanf ("%d", &n)  ;
   switch (n)
   {
      case 0 : printf     ("nul\n");
      case 1 : printf     ("un\n");
      case 2 : printf     ("deux\n");
   }
   printf ("au revoir\n");
}
```

Exemples d'exécution de ce programme

```
donnez un entier : 0
nul
un
deux
au revoir

donnez un entier : 2
deux
au revoir
```

c) l'étiquette "default"

II est possible d'utiliser le mot clé "default" comme étiquette à laquelle le programme se "branchera" dans le cas où aucune valeur satisfaisante n'aura été rencontrée auparavant En voici un exemple :

```
#include<stdio.h>
main()
{
    int n ;
    printf ("donnez un entier : ") ;
    scanf ("%d", &n) ;

    switch (n)
    {
    case 0 : printf ("nul\n") ; break ;
    case 1 : printf ("un\n") ;  break ;
    case 2 : printf ("deux\n"); break ;
    default : printf ("grand\n") ;
    }
    printf ("au revoir\n") ;
}
```

Exemples d'execution de ce programme

```
donnez un entier : 2
deux
au revoir

donnez un entier : 25
grand
au revoir
```

4.2.3. Exercice d'application de switch.

1. Ecrire le programme qui permet de lire un entier compris entre 1 et 7, désignant les jours de la semaine, et affiche le jour correspondant au chiffre saisi, sachant que 1 correspond à lundi.
2. Ecrire un programme C qui permet d'effectuer une opération arithmétique sur deux nombres saisis par l'utilisateur. Le type d'opération doit être indiqué par l'utilisateur.
3. Ecrire le programme qui permet à l'utilisateur de taper un caractère et indique si le caractère taper est un nombre ou la touche *Escape (Esc: Echap)* ou la touche *Entré.*

Touche du clavier	**code ASCII en Décimale**
Entré	10
Esc	27
0	48
1	49
2	50
3	51
4	52
5	53
6	54
7	55
8	56
9	57

4.3. L'INSTRUCTION do... while

La première façon de réaliser une boucle (une itération) en C, à savoir l'instruction *do... while*.

4.3.1. Syntaxe de l'instruction do... while

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

L'instruction est exécutée puis l'expression est évaluée. L'instruction do..while répète l'exécution de *l'instruction* qu'elle contient tant que la valeur de la condition exprimer dans *(expression)* est vraie.

4.3.2. Exemple d'utilisation de l'instruction do..while

```
#include<stdio.h>
main()
{
   int n ;
   do
   {
      printf ("donnez un nb >0 : ") ;
      scanf ("%d", &n) ;
      printf ("vous avez fourni %d\n", n) ;
   }
   white (n<=0) ;
   printf ("réponse correcte") ;
}
```

Exemples d'execution de ce programme

```
donnez un nb >0 : -3
vous avez fourni -3
donnez un nb >0 : -9
vous avez fourni -9
donnez un nb >0 : 12
vous avez fourni 12
réponse correcte
```

On ne sait pas a priori combien de fois une telle boucle sera répétée. Toutefois, de par sa nature même, elle est toujours parcourue au moins une fois. En effet, la condition qui régit cette boucle n'est examinée qu'à la fin de chaque.

4.3.3. Exercice d'application de l'instruction "do..while"

1. Ecrire le programme qui permet à l'utilisateur la saisie d'une seule phrase, et calcul le nombre de caractères saisis, sachant que la fin de la phrase est marquée par un point.
2. Ecrire un programme qui calcul la somme de 10 nombres flottants.
3. Ecrire un programme donne la valeur maximale parmi 10 nombres flottants saisie en utilisant *do..while*.

4.4. L'INSTRUCTION while

la deuxième façon de réaliser une boucle conditionnelle, à savoir l'instruction *while*.

4.4.1. Syntaxe de l'instruction while

```
while (expression)
instruction;
```

L'expression est évaluée. Si sa valeur est VRAIE , l'instruction est alors exécutée. Puis l'expression est a nouveau évaluée.

L'instruction *while* répète alors l'instruction qui la suit tant que la condition mentionnée dans `(expression)` est vraie. comme le ferait *do... while*. Par contre, cette fois, la condition de poursuite est examinée avant chaque parcours de la boucle et non après. Ainsi, contrairement à ce qui se passait avec *do... while*, une telle boucle peut très bien n'être parcourue aucune fois si la condition est fausse dès qu'on l'aborde.

4.4.2. Exemple d'utilisation de l'instruction while

Ce programme permet de répéter la saisie de nombres entiers tant que leur somme est inférieure à 100.

```
#include<stdio.h>
main()
{
   int n, som ; som = 0 ;
   while (som<100)
    {
         printf ("donnez un nombre : ") ;
          scanf ("%d", &n) ;
          som += n ;
   }
   printf ("somme obtenue : %d", som) ;
}
```

Exemples d'execution de ce programme

```
donnez un nombre : 15
donnez un nombre : 25
donnez un nombre : 12
donnez un nombre : 60
somme obtenue : 112
```

4.4.3. Exercice application de l'instruction "while"

1. Ecrire le programme qui permet à l'utilisateur la saisie d'une seule phrase, et calcul le nombre de caractères saisis, sachant que la fin de la phrase est marquée par un point.

2. Ecrire un programme qui calcul la somme de 10 nombres flottants.

3. Ecrire un programme qui donne la valeur maximale parmi 10 nombres flottants saisie par en utilisant *while*.

4.5. L'INSTRUCTION FOR

Etudions maintenant la dernière instruction permettant de réaliser des boucles, à savoir l'instruction ***for***.

4.5.1. Syntaxe de l'instruction for

```
for ([expr1]; [expr2]; [expr3])
          corps-de-boucle
```

L'expression *d'initialisation*: ***expr1*** est évaluée une seule fois, au début de l'exécution de la boucle.

L'expression de test *d'arrêt*: ***expr2*** est évaluée et testée avant chaque passage dans la boucle, si elle est VRAIE le "`corps-de-boucle`" est exécutée.

L'expression *d'incrémentation*: ***expr3*** est évaluée après chaque passage.

N.B. : les crochets ([et]) signifient que ce qu'ils renferment est facultatif.

Formellement ceci est équivalent à:

```
expr1;
while   (expr2)
{
corps-de-boucle;
expr3;
}
```

4.5.2. Exemple d'introduction de l'instruction for

Considérez ce programme :

```
#include<stdio.h>
main()
{
     int i ;
     for ( i=1 ; i<=5 ; i++ )
     {
        printf ("bonjour ") ;
        printf ("%d fois\n", i) ;
     }
}
```

Exemples d'execution de ce programme

bonjour 1 fois
bonjour 2 fois
bonjour 3 fois
bonjour 4 fois
bonjour 5 fois

4.5.3. Exercices d'application de l'instruction "for"

1. Récrire le programme de l'exemple 4.5.2 en utilisant l'instruction *while*.
2. Ecrire un programme qui calcul la somme de 10 nombres flottants en utilisant l'instruction *for*.
3. Ecrire un programme qui donne la valeur maximale parmi 10 nombres flottants saisie par en utilisant *for*.

4.6. LES INSTRUCTIONS DE BRANCHEMENT INCONDITIONNEL : BREAK, CONTINUE ET GOTO

Ces trois instructions fournissent des possibilités diverses de branchement inconditionnel Les deux premières s'emploient principalement au sein de boucles tandis que la dernière est d'un usage libre mais peu répandu, à partir du moment où l'on cherche à structurer quelque peu ses programmes.

4.6.1. L'instruction break

Nous avons déjà vu le rôle de *break* au sein du bloc régi par une instruction *switch*.

Le langage C autorise également l'emploi de cette instruction dans une boucle. Dans ce cas, elle sert à interrompre le déroulement de la boucle, en passant à l'instruction qui suit cette boucle. Bien entendu, cette instruction n'a d'intérêt que si son exécution est conditionnée par un choix ; dans le cas contraire, en effet, elle serait exécutée dès le premier tour de boucle, ce qui rendrait la boucle inutile.

Voici un exemple montrant le fonctionnement de break :

```
main()
{
  int i ;
  for(i=1 ; i<=10 ; i++)
  {
     printf("début tour %d\n", i);
     printf("bonjour\n") ;
     if (i==3) break;
     printf("fin tour %d\n", i);
  }
  printf("après la boucle");
}
```

```
début tour 1
bonjour
fin tour 1
début tour 2
bonjour
fin tour 2
début tour 3
bonjour
après la boucle
```

Remarque :

> *En cas de boucles "imbriquées", break fait sortir de la boucle la plus interne. De même si break apparaît dans un switch imbriqué dans une boucle, elle ne fait sortir que du switch.*

4.6.2. L'instruction continue

L'instruction *continue*, quant à elle, permet de passer "prématurément" au tour de boucle suivant. En voici un premier exemple avec for :

```
main()
{
  int i;
  for (i=1; i<=5; i++)
  {
     printf ("début tour %d\n", i);
     if (i<4) continue ;
     printf ("bonjour\n");
  }
}
```

```
début tour 1
début tour 2
début tour 3
début tour 4
bonjour
début tour 5
bonjour
```

voici un second exemple avec do... while :

```
main();
{
  int n ;
  do
  {
     printf ("donnez un nb>0 : ") ;
     scanf ("%d", &n) ;
     if (n<0)
     {
          printf ("svp >0\n") ;
          continue ;
     }
     printf ("son carré est : %d\n", n*n);
   }
   while(n);
}
```

```
donnez un nb>0 : 4
son carré est : 16
donnez un nb>0 : -5
svp >0
donnez un nb>0 : 2
son carré est : 4
donnez un nb>0 : 0
son carré est : 0
```

Remarques :

1. *Lorsqu'elle est utilisée dans une boucle for, cette instruction continue effectue bien un branchement sur l'évaluation de l'expression de fin de parcours de boucle (nommée expression_2 dans la présentation de sa syntaxe), et non après.*

2. *En cas de boucles "imbriquées", l'instruction continue ne concerne que la boucle la plus interne.*

4.6.3. L'instruction goto

Elle permet "classiquement" le "branchement" en un emplacement quelconque de programme. Voyez cet exemple qui "simule", dans une boucle *for*, l'instruction break à l'aide de l'instruction *goto* (ce programme fournit les mêmes résultats que celui présenté comme exemple de l'instruction break).

```
main()
{   int  i;
    for (i=1; i<=10; i++)
    {
          printf("début tour %d\n",i);
          printf ("bonjour\n") ;
          if (i==3) goto sortie;
          printf("fin tour %d\n", i);
    }
    sortie : printf("après la boucle");
}
```

```
début tour 1
bonjour
fin tour 1
début tour 2
bonjour
fin tour 2
début tour 3
bonjour
après la boucle
```

4.6.4. Exercices

Exercice 1 :

Soit le petit programme suivant :

```
#include <stdio.h>
main();
{
       int i, n, som;
       som = 0;
       for(i=0; i<4; i++)
       {
              printf("donnez un entier ");
              scanf("%d", &n);
              som+=n;
       }
       printf("Somme : %d\n", som);
}
```

Ecrire un programme réalisant exactement la même chose, en employant, à la place de l'instruction for :

1. une instruction while,
2. une instruction do... while.

Exercice 2 :

Calculer la moyenne de notes fournies au clavier avec un "dialogue" de ce type :

```
note 1  : 12
note 2  : 15.25
note 3  : 13.5
note 4  : 8.75
note 5  : -1
moyenne de ces 4 notes : 12.37
```

le nombre de notes n'est pas connu a priori et l'utilisateur peut en fournir autant qu'il le désire. Pour signaler qu'il a terminé, on convient qu'il fournira une note fictive négative. (celle-ci ne devra naturellement pas être prise en compte dans le calcul de la moyenne.

Exercice 3 :

Afficher un triangle rempli d'étoiles, s'étendant sur un nombre de lignes fourni en donnée et se présentant comme dans cet exemple :

```
*
**
***
****
*****
```

Exercice 4 :

Déterminer si un nombre entier fourni en donnée est premier ou non.

Exercice 5 :

Ecrire un programme qui détermine la n-ième valeur u_n (n étant fourni en donnée) de la "suite de Fibonacci" définie comme suit :

$u_1 = 1$

$u_2 = 1$

$u_n = u_{n-1} + u_{n-2}$ Pour n>2

Exercice 6 :

Ecrire un programme qui affiche la "table de multiplication" des nombres de 1 à 10, sous la forme suivante :

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

5. Les tableaux et les pointeurs

Comme tous les langages, C permet d'utiliser des "***tableaux***". On nomme ainsi un ensemble d'éléments de même type désignés par un identificateur unique ; chaque élément est repéré par un "indice" précisant sa position au sein de l'ensemble.

Par ailleurs, comme certains langages tels que Pascal, le langage C dispose de "***pointeurs***", c'est-à-dire de variables destinées à contenir des adresses d'autres "objets" (variables, fonctions...).

A priori, ces deux notions de tableaux et de pointeurs peuvent paraître fort éloignées l'une de l'autre. Toutefois, il se trouve qu'en C un lien indirect existe entre ces deux notions, à savoir qu'un identificateur de tableau est une "constante pointeur". Cela peut se répercuter dans le traitement des tableaux, notamment lorsque ceux-ci sont transmis en argument de l'appel d'une fonction.

C'est ce qui justifie que ces deux notions soient regroupées dans un seul chapitre.

5.1. les tableaux à une dimension

5.1.1. exemple d'utilisation d'un tableau en C

Supposons que nous souhaitions déterminer, à partir de vingt notes d'élèves (fournies en données), combien d'entre elles sont supérieures à la moyenne de la classe.

S'il ne s'agissait que de calculer simplement la moyenne de ces notes, il nous suffirait d'en calculer la somme, en les cumulant dans une variable, au fur et à mesure de leur lecture Mais, ici, il nous faut à nouveau pouvoir consulter les notes pour déterminer combien d'entre elles sont supérieures à la moyenne ainsi obtenue. Il est donc nécessaire de pouvoir "mémoriser" ces vingt notes.

Pour ce faire, il paraît peu raisonnable de prévoir vingt variables scalaires différentes (méthode qui, de toute manière, serait difficilement appliquer à un nombre important de notes).

Le **tableau** va nous offrir une solution convenable à ce problème, comme le montre le programme suivant.

```
#include <stdio.h>
main()
{
      int i, som, nbm ;
      float moy;
      int t[20];
      for (i=0 ; i<20 ; i++)
      {
        printf("donnez la note numéro %d :", i+1) ;
        scanf("%d", &t[i]) ;
      }
      For(i=0, som=0 ; i<20 ; i++)  som += t[i] ;
      moy = som /20 ;
      printf("\n\n moyenne de ta classe : %f\n", moy);
      for(i=0, nbm=0 ; i<20 ; i++ )
          if(t[i]> moy) nbm++ ;
      printf("%d élèves ont plus de cette moyenne", nbm);
}
```

5.1.2. syntaxe

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

`type` : tout type sauf le type tableau (int, float, char…)
`nom tableau` : identificateur de la variable tableau (dans l'exemple précédent c'est **`t`**)
`taille` : nombre d'elements que constituants le tableau (dans l'exemple précédent c'est **`20`**)

Dans l'exemple précédent on a déclaré un tableau nommé *t* de type *int* constitué de 20 éléments. Chaque élément est repéré par sa "position" dans le tableau, nommée "indice". Conventionnellement, en langage C, la première position porte le numéro 0. Ici, donc, nos indices vont de 0 à 19. Le premier élément du tableau sera désigné par t[0], le troisième par t[2], le dernier par t[19].

Plus généralement, une notation telle que t[i] désigne le $(i+1)^{éme}$ élément du tableau.

La notation &t[i] désigne l'adresse de cet élément t[i] de même que &n désignait l'adresse de n.

5.1.3. quelques règles

a) Les éléments de tableau

- *affectation:*

  on peut affecter n'importe quelle valeur à n'importe quel élément du tableau, à condition qu'elle soit de même type que celui du tableau.

```
t[2] = 5;
```

- *incrémentation :* un élément du tableau peut apparaître comme opérande d'un opérateur d'incrémentation, comme dans :

```
t[3]++          --t[i]
```

b) Les indices

Un indice peut prendre la forme de n'importe quelle expression arithmétique de type entier (ou caractère, compte tenu des règles de conversion systématique). Par exemple, si n, p, k et j sont de type *int*, ces notations sont correctes :

```
t[n-3]
t[3*p-2*k+j%l]
```

il en va de même, si cl et c2 sont de type *char*, de :

```
t[c1+3]
t[c2-cl]
```

c) La dimension d'un tableau

La dimension d'un tableau (son nombre d'éléments) ne peut être qu'une **constante** ou une **expression constante.** Ainsi, cette construction :

```
#define N 50
........
int t[N];
float h[2*N-1]   ;
```

est correcte.

5.2. les tableaux à plusieurs dimensions

Comme tous les langages, le langage C autorise les tableaux à plusieurs indices (on dit aussi à plusieurs dimensions).

5.2.1. syntaxe

```
type    nom_tableau[taille_1][taille_2]... [taille_n];
```

`type` : tout type sauf le type tableau.
`nom_tableau` : identificateur de la variable tableau.
`taille 1` : taille de la première dimension
`taille 2` : taille de la deuxième dimension
`...`
`taille n` : taille de la deuxième dimension

Exemples de déclaration :

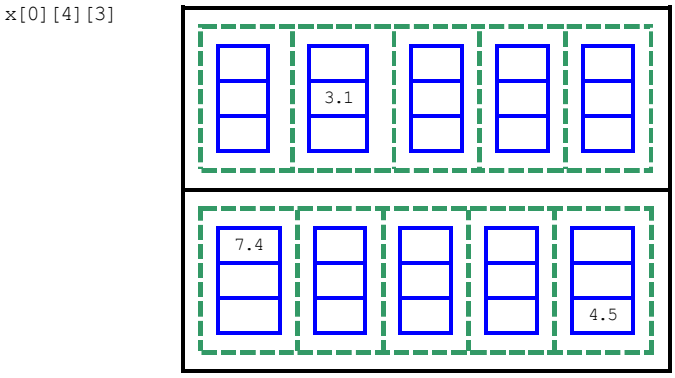
```
1.      int     tab[5][3];
```

réserve un tableau de 15 (5 x 3) éléments. Un élément quelconque de ce tableau se trouve alors repéré par deux indices comme dans ces notations :

```
tab[3][2]      tab[i][j]        tab[i-2][j+i]
```

```
2.      float  x[2][5][3];
```

réserve un tableau de 60 (2 x 5 x 6) éléments. Un élément quelconque de ce tableau se trouve alors repéré par deux indices comme dans ces notations :



5.2.2. Arrangement en mémoire des tableaux à plusieurs indices

Les éléments d'un tableau sont rangés suivant l'ordre obtenu en faisant varier le dernier indice en premier. Ainsi, le tableau t déclaré précédemment dans l'exemple 1. verrait ses éléments ordonnés comme suit :

```
tab[0][0]
tab[0][1]
tab[0][2]
tab[1][0]
tab[1][1]
tab[1][2]
.....
tab[4][0]
tab[4][1]
tab[4][2]
```

5.3. Initialisation des tableaux

Comme les variables scalaires, les tableaux peuvent être initialiser, suivant leur déclaration, de classe statique ou automatique. Les tableaux de classe statique sont, par défaut, initialisés à zéro ; les tableaux de classe automatique ne sont pas initialisés implicitement.

Il est possible, comme on le fait pour une variable scalaire, d'initialiser (partiellement ou totalement) un tableau lors de sa déclaration. Cette fois, cependant, les valeurs fournies devront obligatoirement être des expressions constantes, et cela quelle que soit la classe d'allocation du tableau concerné (alors que les variables scalaires automatiques pouvaient être initialisées avec des expressions quelconques).

Voici quelques exemples vous montrant comment initialiser un tableau.

5.3.1. Initialisation de tableaux à un indice (à une dimension)

Exemple de déclaration avec initialisation:

`int tab [5] = {10, 20, 5, 0, 3 } ;` place les valeurs 10, 20, 5, 0 et 3 dans chacun des cinq éléments du tableau *tab.*

Il est possible de ne mentionner dans les accolades que les premières valeurs, comme dans ces exemples :

```
int tab [5] = {10, 20} ;
int tab[5] = {10, 20, 5} ;
```

Les valeurs manquantes seront, suivant la classe d'allocation du tableau, initialisées à zéro (statique) ou aléatoires (automatique).

De plus, il est possible d'omettre la dimension du tableau, celle-ci étant alors déterminée par le compilateur par le nombre de valeurs énumérées dans l'initialisation. Ainsi, la première déclaration de ce paragraphe est équivalente à :

```
int tab[] = { 10, 20, 5, 0, 3 } ;
```

5.3.2. Initialisation de tableaux à plusieurs indices

Voyez ces deux exemples équivalents (nous avons volontairement choisi des valeurs consécutives pour qu'il soit plus facile de comparer les deux formulations) :

`int tab[3][4] = {{ 1, 2, 3, 4 },`
`{ 5, 6, 7, 8 },`
`{ 9, 10, 11, 12 } };`
	⇔	`int tab [3][4] = {{ 1, 2, 3, 4 },{ 5, 6, 7, 8 },{ 9, 10, 11, 12 } };`

`int tab[3][4] = {1, 2, 3, 4 , 5, 6, 7, 8 , 9, 10, 11, 12 };`		

La première forme revient à considérer notre tableau comme formé de trois tableaux de quatre éléments chacun. La seconde, elle, exploite la manière dont les éléments sont effectivement rangés en mémoire et elle se contente d'énumérer les valeurs du tableau suivant cet ordre.

Là encore, à chacun des deux niveaux, les dernières valeurs peuvent être omises. Les déclarations suivantes sont correctes (mais non équivalentes) :

```
int tab[3][4] = { { 1, 2 } , { 3, 4, 5 } } ;
int tab[3][4] = { 1, 2 , 3, 4, 5 } ;
```

5.4. Exercices

1) Ecrire un programme qui lit 10 nombres entiers dans un tableau, puis les affiches les uns après les autres.
2) Ecrire un programme qui lit 10 nombres entiers dans un tableau avant d'en rechercher le plus grand et le plus petit.
3) Ecrire un programme permettant de trier par ordre croissant les valeurs entières d'un tableau de taille 10. Le tri pourra se faire par réarrangement des valeurs au sein du tableau lui-même.
4) Ecrire un programme calculant la somme de deux matrices de dimension (2,3) dont les éléments sont de type double.

*5.5. Notion de pointeur, les opérateurs * et &*

5.5.1. introduction

Nous avons déjà été amené à utiliser l'opérateur & pour désigner l'adresse d'une variable. D'une manière générale, le langage C permet de manipuler des adresses par l'intermédiaire de variables nommées "pointeurs".

En guise d'introduction à cette nouvelle notion, considérons les instructions :

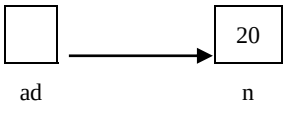
```
int *ad ;
int n ;
n = 20 ;
ad = &n ;
*ad = 30 ;
```

La première réserve une variable nommée *ad* comme étant un "pointeur" sur des entiers. Nous verrons que * est un opérateur qui désigne le contenu de l'adresse qui le suit. Ainsi, à titre "mnémonique", on peut dire que cette déclaration signifie que **ad*, c'est-à-dire l'objet d'adresse *ad*, est de type *int* ; ce qui signifie bien que *ad* est l'adresse d'un entier.

L'instruction :

```
ad = &n ;
```

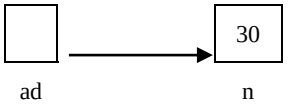
affecte à la variable *ad* la valeur de l'expression *&n*. L'opérateur & (que nous avons déjà utilisé avec *scanf)* est un opérateur unaire qui fournit comme résultat l'adresse de son opérande n. Ainsi, cette instruction place dans la variable *ad* l'adresse de la variable n. Après son exécution, on peut schématiser ainsi la situation :



L'instruction suivante :

```
*ad = 30 ;
```

Signifie : affecter à la variable **ad* la valeur 30. Or **ad* représente l'entier ayant pour adresse *ad* (notez bien que nous disons "l'entier" et pas simplement la "valeur" car, ne l'oubliez pas, *ad* est un pointeur sur des entiers). Après exécution de cette instruction, la situation est la suivante :



Bien entendu, ici, nous aurions obtenu le même résultat avec : n = 30;

5.5.2. Quelques exemples

Voici quelques exemples d'utilisation de ces deux opérateurs. Supposez que nous ayons effectué ces déclarations :

```
int *ad1, *ad2, *ad ;
int n = 10, p = 20 ;
```

Les variables *ad1*, *ad2* et *ad* sont donc des pointeurs sur des entiers. Remarquez bien la forme de la déclaration, en particulier, si l'on avait écrit :

```
int *ad1,  ad2,  ad ;
```

la variable *ad1* aurait bien été un pointeur sur un entier (puisque *ad1 est entier) mais *ad2* et *ad* auraient été, quant à eux, des entiers.

Considérons maintenant ces instructions :

```
ad1 = &n ;
ad2 = &p ;
*ad1 = *ad2 + 2 ;
```

Les deux premières placent dans *ad1* et *ad2* les adresses de n et p. La troisième affecte à **ad1* la valeur de l'expression : `*ad2 + 2`

Autrement dit, elle place à l'adresse désignée par *ad1* la valeur (entière) d'adresse *ad2*, augmentée de 2. Cette instruction joue donc ici le même rôle que :

```
n = p+2;
```

De manière comparable, l'expression : `* adl += 3` jouerait le même rôle que : `n = n +3;`

Et l'expression : `(*ad1)++` jouerait le même rôle que `n++` (nous verrons plus loin que, sans les parenthèses, cette expression aurait une signification différente).

5.5.3. Incrémentation de pointeurs

Jusqu'ici, nous nous sommes contenté de manipuler, non pas les variables pointeurs elles-mêmes, mais les valeurs pointées. Or si une variable pointeur *ad* a été déclarée ainsi :

```
int *ad;
```

une expression telle que : `ad+1` a un sens pour C.

En effet, *ad* est censée contenir l'adresse d'un entier et, pour C, l'expression ci-dessus représente **l'adresse de l'entier suivant.** Certes, dans notre exemple, cela n'a guère d'intérêt car nous ne savons pas avec certitude ce qui se trouve à cet endroit. Mais nous verrons que cela s'avérera fort utile dans le traitement de tableaux ou de chaînes.
Notez bien qu'il ne faut pas confondre un pointeur avec un nombre entier. En effet, l'expression ci-dessus ne représente pas l'adresse de *ad* augmentée de un (octet). Plus précisément**,** la différence entre *ad+1* et *ad* est ici de *sizeof(int)* octets. Si *ad* avait été déclarée par:

```
double *ad;
```

cette différence serait de *sizeof(double)* octets. De manière comparable, l'expression :

`ad++` incrémente l'adresse contenue dans *ad* de manière qu'elle désigne ***l'objet*** suivant.

Notez bien qu'il est possible d'incrémenter ou de décrémenter un pointeur de n'importe quelle quantité entière. Par exemple, avec la déclaration précédente de ad, nous pourrions écrire ces instructions :

```
ad += 10 ;
ad -= 25 ;
```

5.5.4. Un nom de tableau est un pointeur constant

En langage C, l'identificateur d'un tableau, lorsqu'il est employé seul (sans indices à sa suite), est considéré comme un pointeur (constant) sur le début du tableau. Nous allons en examiner les conséquences en commençant par le cas des tableaux à un indice ; nous verrons en effet que, pour les tableaux à plusieurs indices, il faudra tenir compte du type exact du pointeur en question.

a) Cas des tableaux à un indice

Supposons, par exemple, que l'on effectue la déclaration suivante :

```
int     t[10];
```

la notation `t` est alors totalement équivalente à `&t[0]`.

L'identificateur `t` est considéré comme étant de type pointeur sur le type correspondant aux éléments du tableau, c'est-à-dire, ici, int *. Ainsi, voici quelques exemples de notations équivalentes :

```
t+1     ⇔  &t[1]
t+i     ⇔  &t[i]
t[i]    ⇔  *(t+i)
```

Pour illustrer ces nouvelles possibilités de notation, voici plusieurs façons de placer la valeur 1 dans chacun des 10 éléments de notre tableau t :

```
int i ;
for(i=0 ; i<10; i++)
    (t+i) = 1;
```

⇔

```
int i ;
int * p;
for(p=t, i=0 ; i<10; i++, p++)
     *P = 1;
```

Dans la seconde façon, nous avons dû recopier la "valeur" représentée par t dans un pointeur nommé p. En effet, il ne faut pas perdre de vue que le symbole t représente une adresse constante (t est une constante de type pointeur sur des entiers). Autrement dit, une expression telle que `t++` aurait été invalide, au même titre que, par exemple, `3++`. Un nom de tableau est un pointeur constant ; ce n'est pas une *variable*.

Remarque importante :

Nous venons de voir que la notation `t[i]` est équivalente à `*(t+i)` lorsque t est déclaré comme un tableau. En fait, cela reste vrai, quelle que soit la manière dont t a été déclaré. Ainsi, avec :

```
int * t ;
```

Les deux notations précédentes resteraient équivalentes. Autrement dit, on peut utiliser `t[i]` dans un programme où `t` est simplement déclaré comme un pointeur (encore faudra-t-il, toutefois, avoir alloué l'espace mémoire nécessaire).

b) Cas des tableaux à plusieurs indices

Comme pour les tableaux à un indice, l'identificateur d'un tableau, employé seul, représente toujours son "adresse" de début. Toutefois, si l'on s'intéresse à son type exact, il ne s'agit plus d'un pointeur sur des éléments du tableau. En pratique, ce point n'a d'importance que lorsque l'on effectue des calculs arithmétiques avec ce pointeur (ce qui est assez rare) ou lorsque l'on doit transmettre ce pointeur en argument d'une fonction

A simple titre indicatif, nous vous présentons ici les règles employées par C, en nous limitant au cas de tableaux à deux indices.

Lorsque le compilateur rencontre une déclaration telle que : `int  t[3][4];`

il considère en fait que `t` désigne un tableau de 3 éléments, chacun de ces éléments étant lui-même un tableau de 4 entiers. Autrement dit, si `t` représente bien l'adresse de début de notre tableau `t`, il n'est plus de type `int *` (comme c'était le cas pour un tableau à un indice) mais d'un type "`pointeur sur des blocs de 4 entiers`", type qui devrait se noter théoriquement4 :

```
int [4] *
```

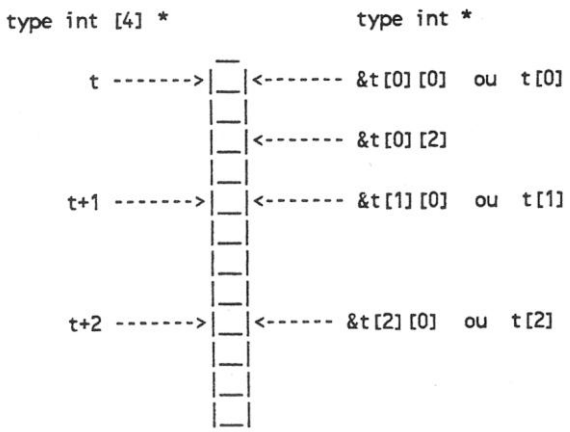
Dans ces conditions, une expression telle que `t+1` correspond à l'adresse de `t`, augmentée de 4 entiers (et non plus d'un seul !). Ainsi, les notations `t` et `&t[0][0]` correspondent toujours à la même adresse, mais l'incrémentation de `1` n'a pas la même signification pour les deux.

D'autre part, les notations telles que `t[0]`, `t[1]` ou `t[i]` ont un sens. Par exemple, `t[0]` représente l'adresse de début du premier bloc (de 4 entiers) de `t`; `t[1]` celle du second bloc... Cette fois, il s'agit bien de pointeurs de type `int *`. Autrement, dit les notations suivantes sont totalement équivalentes (elles correspondent à la même adresse et elles sont de même type) :

`t[0]` ⇔ `&t[0][0]`

`t[1]` ⇔ `&t[1][0]`

Voici un schéma récapitulant ce que nous venons de dire.



Remarque :

`t[1]` est une *constante* ; ce n'est pas une *variable.* L'expression : `t[1]++` est invalide. Par contre, `t[1][2]` est bien une *variable.*

5.5.5. Les opérations réalisables sur des pointeurs

Nous avons déjà vu ce qu'étaient la somme ou la différence d'un pointeur et d'une valeur entière. Nous allons examiner ici les autres opérations qu'on peut réaliser avec des pointeurs.

a) La comparaison de pointeurs

II ne faut pas oublier qu'en C un pointeur est défini à la fois par une *adresse* en mémoire et par un *type*. On ne pourra donc comparer que des pointeurs de même *type*. Par exemple, voici, en parallèle, deux suites d'instructions réalisant la même action : mise à 1 des 10 éléments du tableau *t* :

```
int t[10];
int *p ;
for (p=t ; p<t+10 ; p++)
    *p = 1 ;
```

```
int t[10];
int i;
for (i=0 ; i<10; i++)
     t[i]=1;
```

b) soustraction de pointeurs

Là encore, quand deux pointeurs sont de même type, leur différence fournit le nombre d'éléments du type en question situés entre les deux adresses correspondantes. L'emploi de cette possibilité est assez rare.

c) Les affectations de pointeurs et le pointeur nul

Nous avons naturellement déjà rencontré des cas d'affectation de la valeur d'un pointeur à un pointeur de même type. A priori, c'est le seul cas autorisé par le langage C (du moins, tant que l'on ne procède pas à des conversions explicites). Une exception a toutefois lieu en ce qui concerne la "valeur entière 0", Bien entendu, cela n'a d'intérêt que parce qu'il est possible de comparer n'importe quel pointeur (de n'importe quel type) avec ce "pointeur nul".

D'une manière générale, plutôt que la valeur 0, il est conseillé d'employer la constante ***NULL*** prédéfinie dans *stdio.h* (bien entendu, elle sera remplacée par la constante entière 0 lors du traitement par le préprocesseur, mais les programmes source en seront néanmoins plus lisibles).

Avec ces déclarations :

```
int     *n ;
double  *x ;
```

Ces instructions seront correctes :

```
n = 0 ;          /* ou mieux */          n = NULL ;
x = 0 ;          /* ou mieux */          x = NULL ;
if (n == 0) ...  /* ou mieux */          if (n == NULL) ...
```

5.6. Exercices

1) Ecrire un programme qui lit N nombres entiers, puis les affiches les uns après les autres, la valeur N est donnée en entrée.
2) Ecrire un programme calculant la somme de N nombres et leur moyenne puis affiche ces nombres.
3) Ecrire un programme qui lit N nombres entiers, avant d'en rechercher le plus grand et le plus petit.
4) Ecrire un programme utilisant les pointeurs, permettant de trier par ordre croissant les valeurs entières d'un tableau de taille N. Le tri pourra se faire par réarrangement des valeurs au sein du tableau lui-même.

6. Les chaînes de caractères

Certains langages (tels que le Basic ou le Turbo Pascal) disposent d'un véritable "type chaîne". Les variables d'un tel type sont destinées à recevoir des suites de caractères qui peuvent évoluer, à la fois en contenu et en longueur, au fil du déroulement du programme. Elles peuvent être manipulées d'une manière "globale", en ce sens qu'une simple affectation permet de transférer le contenu d'une variable de ce type dans une autre variable de même type.

D'autres langages (tels que le Fortran ou le Pascal "standard") ne disposent pas d'un tel type chaîne. Pour traiter de telles informations, il est alors nécessaire de travailler sur des tableaux de caractères dont la taille est nécessairement fixe (ce qui impose à la fois une longueur maximale aux chaînes et ce qui, du même coup, entraîne une perte de place mémoire). La manipulation de telles informations est obligatoirement réalisée caractère par caractère et il faut, de plus, prévoir le moyen de connaître la longueur courante de chaque chaîne.

En langage C, il n'existe pas de véritable type chaîne, dans la mesure où l'on ne peut pas y déclarer des variables d'un tel type. Par contre, il existe une convention de représentation des chaînes. Celle-ci est utilisée à la fois :

- par le compilateur pour représenter les chaînes constantes (notées entre doubles quotes),
- par un certain nombre de fonctions qui permettent de réaliser :
  - les lectures ou écritures de chaînes,
  - les "traitements classiques" tels que concaténation, recopie, comparaison, extraction de sous-chaîne, conversions,...

Mais, comme il n'existe pas de variables de type chaîne, il faudra prévoir un emplai, n pour accueillir ces informations. Un tableau de caractères pourra faire l'aflhin ' d'ailleurs ce que nous utiliserons dans ce chapitre. Mais nous verrons plus tard i onui créer "dynamiquement" des emplacements mémoire, lesquels seront alors reperr. |>.u pointeurs.

6.1. REPRÉSENTATION DES CHAÎNES

6.1.1. La convention adoptée

En C, une chaîne de caractères est représentée par une suite d'octets correspondit)! chacun de ses caractères (plus précisément à chacun de leurs codes), le toui étant ttfflj par un octet supplémentaire de code nul. Cela signifie que, d'une manière géndnli u chaîne de n caractères occupe en mémoire un emplacement de n+1 octets.

6.1.2. Cas des chaînes constantes

C'est cette convention qu'utilise le compilateur pour représenter les "constantes ciinlii (sous-entendu que vous introduisez dans vos programmes), sous des notations <lr forme :

"bonjour"

De plus, une telle notation sera traduite par le compilateur en un pointeur (sur éléments de type char) sur la zone mémoire correspondante.

Voici un programme illustrant ces deux particularités :

```
#include <stdio.h>
main0
f char * adr ;
   adr = "bonjour" ; while (*adr) i printf ("%c", *adr) adr++ ;
```

bon,,,.,.