Algorithmique & Langage C
IUT GEII S1

Les pointeurs
Les structures
Opérations bits à bits
Révisions

Notes de cours

cours_algo_lgc8.06.odp (huitième partie)

Licence

La mémoire

Une mémoire est un circuit à semi-conducteur permettant d'enregistrer, de conserver et de restituer des informations.

Elle est utilisée pour stocker les instructions d'un programme ainsi que les données associées (variables, constantes).

Une mémoire peut être représentée comme une armoire de rangement constituée de différents tiroirs.

Chaque tiroir représente alors une case mémoire qui peut contenir un seul élément : des données.

Chaque tiroir possède un numéro appelé adresse.

Chaque donnée devient alors accessible grâce à son adresse.


La mémoire

0xFFFFFFFF	10011001
0xFFFFFFFE	00001000
0xFFFFFFFD	01111111
0xFFFFFFFC	01111001
0x00000003	01001111
0x00000002	00001011
0x00000001	11001011
0x00000000	01001001

Adresses (ici 32bits) Données (ici 8bits)


La taille des variables (données)

Le seul type dont la taille est fixe est char :

taille d'un char = 1 octet.

Pour les autres types, la taille dépend de l'architecture et du compilateur. Le langage C n'impose que des tailles minimum :

Ex : taille d'un int >= 2 octets.

On peut connaître la taille réelle par l'opérateur **`sizeof`** :

`sizeof ( nom_de_type )` : retourne le nombre d'octets utilisés pour stocker une variable de ce type.

`sizeof( variable )` : retourne le nombre d'octets utilisés pour stocker cette variable.


La taille des variables (données)

Soit les déclarations :

```
char c ; int i ; float f ;
char ch[10] ; int tab[10]; float tabf[10];
```

`size_t` est le type pour les tailles (souvent int) :

```
size_t tc, ti , tf ;

tc = sizeof ( c );   ou tc = sizeof ( char ) ;
ti = sizeof ( i ) ;  ou ti = sizeof ( int ) ;
tf = sizeof ( f ) ;  ou tf = sizeof ( float ) ;

tc = sizeof ( ch ) ;    // 10 x sizeof ( char )
ti  = sizeof ( tab ) ;  // 10 x sizeof ( int )
tf =  sizeof ( tabf ) ; // 10 x sizeof ( tabf )
```

voir tailles.c


Manipuler des adresses en C

En C, on ne manipule généralement pas une adresse sans tenir compte du type des données contenues à cette adresse.

On a plusieurs types "adresse" selon la donnée :

```
char *     : adresse d'un char
int *      : adresse d'un int
float *    : adresse d'un float
.....
```

Un **pointeur** est une variable dont la valeur représente une adresse :

```
int * p ;
char * pc ;
```


Little Endian / Big Endian

Pour stocker une donnée comportant plusieurs octets dans une mémoire dont la taille des données sont des octets, il y a deux manière de procéder :

- **Little Endian** : le poids faible dans l'adresse de plus faible valeur (the little end comes first)

µP : Intel , Microchip, ...

- **Big Endian** : le poids faible dans l'adresse de valeur la plus forte (the big end comes first)

µP : Freescale (Ex Motorola), ...


Little Endian / Big Endian

La valeur entière 109243=0x0001AABB est stockée à partir de l'adresse 0x00221234 :

Little Endian :

0x00221237	0x00
0x00221236	0x01
0x00221235	0xAA
0x00221234	0xBB

int i = 109243 ;

Big Endian :

0x00221237	0xBB
0x00221236	0xAA
0x00221235	0x01
0x00221234	0x00

int i = 109243 ;


Opérateurs * et &

Soit une variable destinée à contenir l'adresse d'un entier :

```
int  *  p ;
```

Soit un entier :

```
int i ;
```

On met dans p l'adresse de i :

```
p  = &i ;
```

Maintenant on dispose de deux moyens d'accéder à la valeur de i :

```
i  = 7 ;
```

ou bien :

```
*p = 7 ;
```

contenu de (p) ← 7


Pointeurs

Déclaration : <type> * <nom_du_pointeur> ;

```
int *  p ;
```

Initialisation : <nom_du_pointeur> = &<variable> ;

```
int i ;
p  =   & i ;
```

Utilisation : *<nom_du_pointeur> = ... ;

```
*p = 7 ;
```


Pointeurs

On peut afficher une adresse avec le format %p :

```
int *p ;
int i ;

printf("Adresse de i : %p\n" , &i ) ;

printf("Valeur de p (quelconque) : %p", p);

p = &i ;

printf("Valeur de p : %p\n", p );

*p = 7 ;

printf("i = %d ,  *p = %d\n", i , *p ) ;
```


Pointeurs et tableaux

```
char [10];  int i ;

char  * p ;
```

initialisation avec l'adresse du 1er élément
le nom d'un tableau est l'adresse du 1er élément

```
p = tab ;

// mettre 'a' dans chaque éléments
i= 0 ;
while (   i < 9 ) {
                    *p = 'a' ;
                     p = p + 1 ;
                     i = i + 1 ;
                  }
*p = '\0' ;
```


Pointeurs et tableaux

L'instruction :

```
p =  p + 1 ;
```

augmente le pointeur de la taille du type pointé.

Ainsi quelque soit la taille du type on passe à l'élément suivant.

```
float *p ;

p = p + 1 ;
```
augmente p de sizeof(float) (ici 4)

```
int k ;

p  = p + k ;
```
p augmente de k fois la taille du type pointé par p


Pointeurs et tableaux

```
char * p ;   char tab[10] ;

 p = tab ;
```

La notation **`*p`** est ici équivalente à **`tab[0]`**

```
  *p = 'a' ;
```
est identique **`tab[0] = 'a' ;`**

On "pointe" maintenant le prochain élément :

```
   p = p + 1 ;
```

Maintenant **`*p`** est équivalente à **`tab[1]`** :

```
    *p = 'a' ;
```
est identique **`tab[1] = 'a' ;`**


Pointeurs et fonctions

Voici une fonction qui retourne le nombre de caractères 'e' dans une chaîne :

Prototype :

```
int nbe ( char * ) ;
```

Définition :

```
int nbe ( char * ch ) {
        int ne = 0  , i = 0 ;
        while ( ch[i]!='\0' ) {
                if ( ch[i] == 'e' ) {
                        ne = ne + 1 ; }
               i = i + 1 ;
               }
        return ne ; }
```

Appel :

```
int n ;
n = nbe("ecologiquement" );
```


Pointeurs et fonctions

On peut écrire la même fonction en utilisant la notation pointeur :

```
int nbe ( char * ch ) {

        int ne = 0 ;

        while ( *ch != '\0' ) {

                if ( *ch == 'e' ) {
                        ne = ne + 1 ; }

                ch = ch + 1 ;

                }

        return ne ; }
```


Pointeurs et fonctions

Comme on passe l'adresse (par un pointeur) la fonction peut modifier la valeur du paramètre :

```
void Remplace_a_par_b ( char * ch ) {

        while ( *ch != '\0' ) {

                if ( *ch == 'a' ) {
                        *ch = 'b' ; }

                ch = ch + 1 ;

                }
}
```


le & dans scanf

La fonction scanf doit avoir accès à l'adresse de la variable pour pouvoir la modifier :

```
int i ;
printf("Entrez un entier : ") ;
scanf ( "%d" , &i ) ;


char ch[80] ;
printf("Entrez un mot : " );
scanf("%s" , ch  ) ;
```

Le nom d'un tableau étant l'adresse de son premier élément, il ne faut pas de & dans ce cas.


Structures

Une structure est un type qui rassemble des données de types différents.

Définition d'une structure :

```
struct Etudiant {
          char nom[20];
          char prenom[20];
          int semestre ;
          char groupe ;
          float moyenne ; } ;
```

Déclaration de variables de type structure étudiant :

```
struct Etudiant Et1 , Et2 ;
```


Structures

On accède aux éléments par l'opérateur . .

```
struct Etudiant Et1 , Et2 ;

strcpy(Et1.nom,"Einstein");
strcpy(Et1.prenom,"Albert");
Et1.groupe = 'A' ;
Et1.semestre = 1 ;

if ( Et1.moyenne >= 10.0 ) {
     Et1.semestre = Et1.semestre + 1 ; }
```


Structures

Tableau de structures :

```
struct Etudiant Promotion[120] ;
int i ;

     for ( i = 0 ; i < 120 ; i++ ) {
                 Promotion[i].semestre = 1 ; }
```

Pointeur sur une structure :

```
struct Etudiant * Pedt ;
struct Etudiant Et1 ;
Pedt = &Edt1 ;
```

dans ce cas on accède aux champs par l'opérateur ->

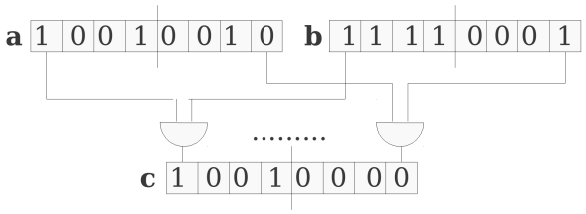
```
Pedt->moyenne = 12.0 ;
```


Opérateur ET bit à bit

Opérateurs ET bit à bit : **&**

(à ne pas confondre avec le ET logique déjà vu : **&&**)

```
char  a = 0x92 , b = 0xF1 , c ;

c = a & b ;
```

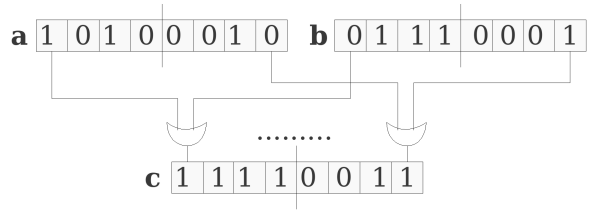


Opérateur OU bit à bit

Opérateurs OU bit à bit : **|**

(à ne pas confondre avec le OU logique déjà vu : **||**)

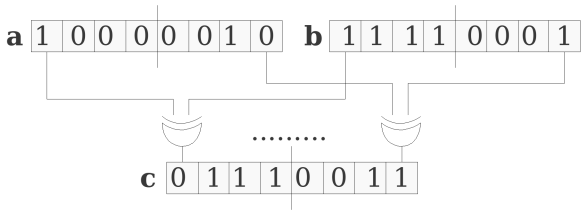
```
char  a = 0xA2 , b = 0x71 , c ;

c = a | b ;
```



Opérateur OU Exclusif bit à bit

Opérateurs OU Exclusif bit à bit : ^

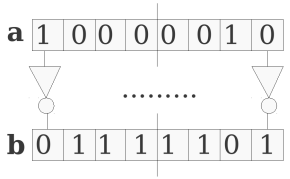
```
char  a = 0x82 , b = 0xF1 , c ;

c = a ^ b ;
```



Opérateur d'inversion bit à bit

Opérateurs d'inversion bit à bit : ~

```
char  a = 0x82 ;

b= ~a ;
```



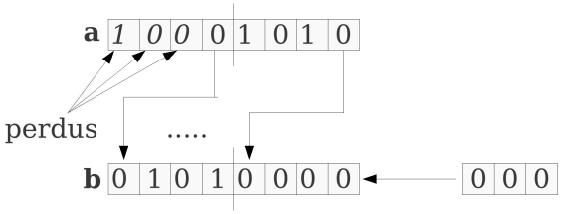
Tous les bits de a sont inversés.
b vaut donc ici 0x7D

Attention à ne pas confondre avec la négation logique **!**
!(x) est vrai (et vaut 1) quand x est faux et inversement.
Comme toute valeur non nulle est vraie, ici avec b=!a;
On aurait b qui vaudrait 0x01.

Opérateur de décalage à gauche

Décalage à gauche : **<<**
Tous les bits sont décalés vers la gauche, les bits qui "entrent" sont des zéros, les bits qui "sortent" sont perdus.

char a=0x8A , b ;
b = a << 3 ; // décalage de trois positions vers la gauche

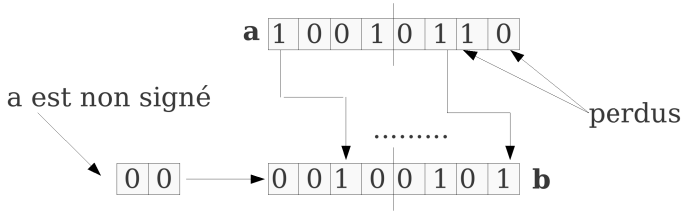


Mathématiquement, un décalage à gauche de n revient à multiplier par 2^n.
Si des bits non nuls sont perdus c'est que le résultat ne tient pas dans le format.

Opérateur de décalage à droite

Décalage à droite : **>>**
Tous les bits sont décalés vers la droite, les bits qui "sortent" sont perdus. Lorsque la valeur est non signée, les bits qui "rentrent" sont égaux à zéro.

unsigned char a=0x96 , b ;
b = a >> 2 ; // décalage de deux positions vers la droite.



Le décalage à droite de n positions correspond à une division par 2^n.
Les bits perdus correspondent à 0.5,0.25,0.125 etc...

Opérateur de décalage à droite

En C, si on décale à droite une valeur signée,
le résultat dépend du compilateur utilisé !
Les bits qui rentrent sont soit des zéros (décalage logique),
soit la valeur du bit de signe (décalage arithmétique).
Il vaut donc mieux éviter d'utiliser l'opérateur >> sur des valeurs signées pour conserver la portabilité.

Ex :
int a = -4 ; // int est un type signé

	1	1	1	1	1	1	0	0	**a**
Dépend du Compilateur !	0	1	1	1	1	1	1	0	**a>>1 logique**
	1	1	1	1	1	1	1	0	**a>>1 arithmétique**


Masque d'un bit

Les masques de bits sont utilisés pour isoler un bit (ou un sous-ensemble de bit). Ce sont des quantités comportant tous les bits à zéros sauf celui (ou ceux) que l'on veut isoler. Les masques permettent de tester ou de modifier des bits sans influencer les autres.

masque du bit 3 : 0000 1000
masque des bits 1 à 4 : 0001 1110

valeur : 0101 1101
masque : 0000 1000

Test d'un bit :
La quantité valeur&masque est non nulle ssi le bit correspondant au masque vaut 1


Tester d'un bit

On fait un ET avec son masque :

Ex : Si le bit 4 de x vaut 1 ....

```
mask = 0x10;   // 0001 0000

if ( x & mask ) {

   // le bit 4 de x vaut 1
} else
{
   // le bit 4 de x vaut 0
}
```


Test d'un bit

En C, le masque du bit n° i peut s'écrire simplement

```
1<<i
```

Donc pour tester le bit i on peut écrire :

```
if ( x & (1<<i) ) {


}
```


Mettre un bit à 1 (sans modifier les autres)

On fait un OU avec le masque du bit

int x ;

x = x | 0x10 ; // met le bit 4 de x à 1

x	1	0	1	0	1	0	0	1
masque	0	0	0	1	0	0	0	0
x | masque	1	0	1	1	1	0	0	1

Rappel de logique
a+1=1
a+0=a

Mettre un bit à 0 (sans modifier les autres)

On fait un ET avec l'inverse du masque du bit

int x ;

x = x & (~0x10) ; // met le bit 4 de x à 0

x	1	0	1	1	1	1	0	1
~masque	1	1	1	0	1	1	1	1
x & ~masque	1	0	1	0	1	1	0	1

Rappel de logique
a.1=a
a.0=0

Inverser un bit (sans modifier les autres)

On fait un Ou exclusif avec le masque du bit

int x ;

x = x ^ 0x10 ; // inverse le bit 4 de x

x	1	0	1	1	1	0	0	1
masque	0	0	0	1	0	0	0	0
x | masque	1	0	1	0	1	0	0	1

Rappel de logique
$a \oplus 1 = \overline{a}$
$a \oplus 0 = a$


Opérateurs "condensés"

En C un expression de la forme

```
x = x op y ;
```

peut s'écrire

```
x op= y ;
```

Pour

```
op ∈ { +, -, * , / , % , &,|, ^,<< ,>> }
```

Ainsi :

```
x = x + 3 ;    <=>  x += 3 ;
y = y - b ;    <=>  y -= b ;
t = z << a ;  <=>  z <<= a ;
t = t & 0x04;  <=> t &= 0x04 ;
etc....
```


Macros de modifications de bits

Il est souvent pratique de définir les macros suivantes pour manipuler les bits :

```
// Mise à un d'un bit d'un octet
#define BIT_SET( octet, nbit )     ( octet |= (1<<(nbit))   )

// Mise à zéro d'un bit dans un octet
#define BIT_CLEAR( octet , nbit )  ( octet &= ~(1<<(nbit))  )

// Test d'un bit dans un octet
// (renvoie 1 si le bit est à 1 et 0 sinon)
#define BIT_TEST( octet , nbit  )  ( octet&(1<<(nbit))? 1:0 )
```


Macros de modifications de bits

Utilisation :

Mettre le bit 4 de c à 1 : `BIT_SET(c,4);`

Mettre le bit 3 de c à 0 : `BIT_CLEAR(c,3);`

Tester le bit 7 de c :

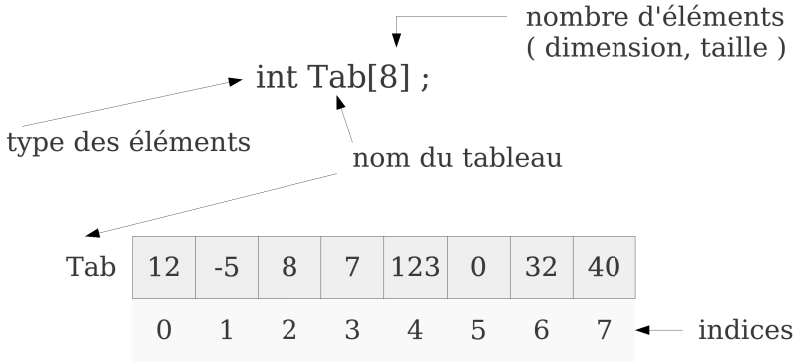
```
if ( BIT_TEST(c,7)==0 ) {
   // b7 de c vaut 0
}
```

ou

```
if ( BIT_TEST(c,7) ) {
  // b7 de c vaut 1
}
```


Révisions


Tableau



ici, l'élément d'indice 4 vaut 123 il est noté **Tab[4]**


Algorithmes à connaître

- Saisir la valeurs des éléments au clavier.
- Afficher la valeurs des éléments.
- Calculer la somme des éléments.
- Recopier les éléments d'un tableau dans un autre.
- Compter le nombre d'éléments qui vérifient une propriété donnée.
- Trouver la plus grande valeur d'un tableau (maximum).


Recopier les éléments d'un tableau dans un autre

```
// recopie des éléments de tab1 dans tab2
int tab1[5] ;
int tab2[5] ;
i=0;
..... ;
while ( i < 5 )
{
tab2[i] = tab1[i] ;
i = i + 1 ;
}
```

taille de tab2 >= taille de tab1

Exercice : Recopier "à l'envers"
(premier -> dernier, deuxième -> avant dernier, ...,...,...,dernier->premier)


Compter le nombre d'éléments qui vérifient une propriété donnée

```
//Compter le nombre d'éléments positifs

int tab[5] ;
int nb_pos = 0 ;
i=0;

while ( i < 5 )
{
  if ( tab[i] > 0 ) {
            nb_pos = nb_pos + 1 ;
          }
i = i + 1 ;
}
```

Ne pas oublier l'initialisation à 0 !


Trouver la plus grande valeur d'un tableau (maximum)

```
int tab[5] ;
int max = tab[0] ;
i = 1;

while ( i < 5 )
{
  if ( tab[i] > max ) {
            max = tab[i] ;
          }
i = i + 1 ;
}
```

max est par définition une valeur du tableau !

Attention : Ne pas initialiser max à 0. Si tous les éléments sont négatifs, on va trouver max égal à 0 !


code ASCII de base (7 bits)

	0	1	2	3	4	5	6	7
0x00	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL
0x10	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB
0x20	SP	!	"	#	$	%	&	'
0x30	0	1	2	3	4	5	6	7
0x40	@	A	B	C	D	E	F	G
0x50	P	Q	R	S	T	U	V	W
0x60	`	a	b	c	d	e	f	g
0x70	p	q	r	s	t	u	v	w

128 caractères de 0x00 à 0x7F

poids fort →

	8	9	A	B	C	D	E	F
0x00	BS	HT	LF	VT	NP	CR	SO	SI
0x10	CAN	EM	SUB	ESC	FS	GS	RS	US
0x20	(	)	*	+	,	-	.	/
0x30	8	9	:	;	<	=	>	?
0x40	H	I	J	K	L	M	N	O
0x50	X	Y	Z	[	\	]	^	_
0x60	h	i	j	k	l	m	n	o
0x70	x	y	z	{	|	}	~	DEL


Quelques propriétés des codes ASCII

Des lettres consécutives dans l'alphabet ont des codes ascii consécutifs :
code de B = (code de A) + 1
code de C = (code de B) + 1 = (code de A) + 2

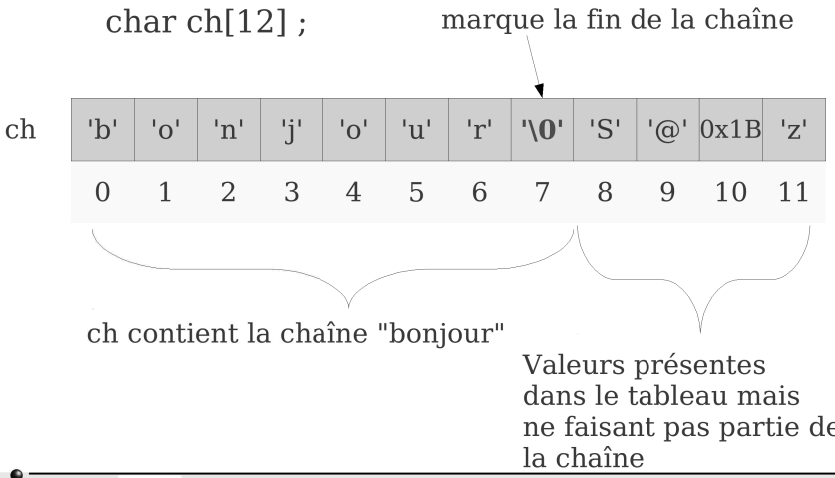
L'écart entre une lettre en minuscule et la même lettre en majuscules est constant et vaut 0x20 :

code de x = code de X + 0x20

Code ascii d'un caractère chiffre=valeur du chiffre + 0x30


Chaîne de caractères

char ch[12] ;



Saisie d'une chaîne

On peut utiliser **scanf** avec le format **%s** :

```
char ch[30] ;

printf("Entrer un mot : ");
scanf("%30s", ch );
```

Avec scanf la saisie de la chaîne s'arrête dès le premier caractères espace (' ') rencontré.

ou bien

```
printf("Entrer un mot : ");
fgets ( ch , 30 , stdin );
```

Ici on peut saisir des chaînes quelconques, mais le '\n' final se retrouve dans la chaîne.

Traitement des chaînes de caractères

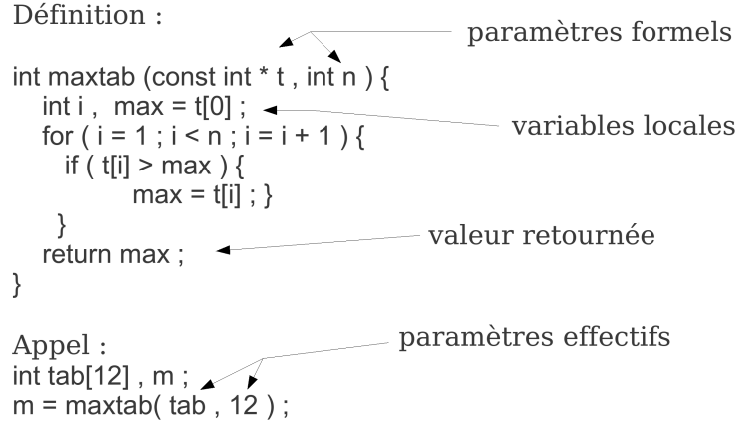
On va utiliser les mêmes méthodes que pour les tableaux quelconques mais au lieu de poursuivre le traitement jusqu'au dernier élément, on va s'arrêter lorsqu'on rencontre la fin de chaîne '\0'.

```
char ch[80];
int i = 0 ;

while ( ch[i] != '\0')
{

   Traiter le caractère n° i ;
i=i+1 ;
}
```


Fonctions

Prototype : int maxtab (const int * t , int n) ;

Définition :



Appel :

Fonctions sur les chaînes de caractères

Une fonction qui retourne le nombre de caractères 'e' dans une chaîne :

Prototype :

```
int nbe ( char * ) ;
```

une chaîne est un tableau de char

Définition :

```
int nbe ( char * ch ) {
        int ne = 0  , i = 0 ;
        while ( ch[i]!='\0' ) {
                if ( ch[i] == 'e' ) {
                        ne = ne + 1 ; }
                i = i + 1 ;
                }
        return ne ; }
```

Appel :

```
int n ;
n = nbe("ecologiquement" );
```


Fonctions sur les chaînes de la librairie standard

La librairie standard du C fournie de nombreuses fonctions bien utiles pour traiter les chaînes de caractères.
Les prototypes se trouvent dans le fichier string.h.

Exemples :

Longueur d'une chaîne :

```
int strlen ( char * ch ) ;
```

Recopie de ch2 dans ch1 :

```
void strcpy ( char * ch1 ,  const char * ch2 ) ;
```

Ajout de ch2 dans ch1 après la fin de ch1 :

```
void strcat ( char * ch1 , const char * ch2 ) ;
```

Comparaison de chaînes :

```
int strcmp ( const char * ch1 , const char * ch2 ) ;
```

retourne 0 si ch1 et ch2 sont identiques.

voir le fichier d'exemple : fct_chaines.c

Écrire dans un fichier

```
FILE * fichier ;

fichier = fopen( "toto.txt" , "w" ) ;

if ( fichier != NULL ) {


        fprintf( fichier , "Bonjour !\n" ) ;
        fclose( fichier );
}
else {
       printf("Erreur d'ouverture du fichier!\n");
     }
```


Lecture caractère par caractère

```
FILE * fichier ;

int c  ;   // déclaration de c en int !!!

fichier = fopen( "toto.txt" , "r" ) ;

if ( fichier != NULL ) {

while( c = fgetc(fichier) , c != EOF ) {


            // exploiter c

      }
     fclose ( fichier ) ;      }
```


Lecture ligne par ligne jusqu'à la fin

```
FILE * fichier ;
float u , i ;
int fin = 0 ;
char ch[160];
fichier = fopen( "mesures.txt" , "r" ) ;
if ( fichier != NULL ) {

while( fin==0 ) {

          fgets( ch , 160 , fichier ) ;
          fin = foef ( fichier ) ;
          if ( fin == 0 ) {
             // exploiter ch
            }
      }

     fclose ( fichier ) ;       }
```