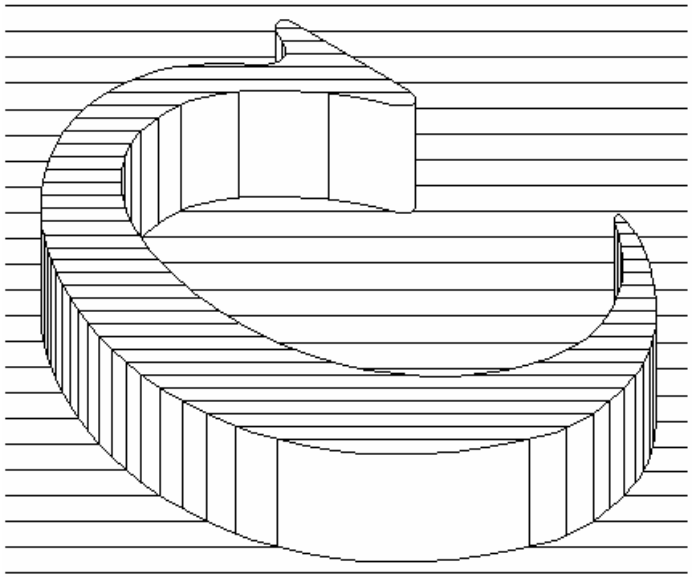
jacqueline.chaboche@laposte.net

Cours de langage C



Sommaire

Index des fonctions de la bibliothèque standard

1. Présentation

1.1. Historique

C a été conçu afin de réécrire de façon portable et en langage évolué UNIX

- **1972** : Dennis Ritchie (Laboratoires Bell, AT&T) crée le langage C
- **1973** : UNIX est réécrit en langage C (D.Ritchie et K.Thompson)
  C s'impose comme le langage de programmation sous UNIX
- **1978** : Kernighan et Ritchie publient une définition du langage C
- **1984** : Harbison et Steele publient une synthèse sur les compilateurs et dialectes
- **1989** : fin du travail de l'ANSI de normalisation du langage, de ses bibliothèques et de son environnement de compilation et d'exécution

C++, développé par B. Stroustrup (*AT&T*), tend à prendre maintenant le relais de C

C++ reste presque compatible avec C ANSI et apporte les concepts des langages orientés objets

1.2. Caractéristiques

langage à usage général

présent sur pratiquement toutes les machines et systèmes d'exploitation

possède des facilités naturelles d'interfaçage avec les primitives système

dispose d'une bibliothèque de fonctions et d'un grand nombre d'opérateurs

structures de contrôle permettant la programmation structurée

architecture bien adaptée à la programmation modulaire

langage relativement dépouillé

sémantiquement près du matériel

les programmes C sont efficaces : le code généré est rapide et compact

très permissif (en particulier, il est faiblement typé)

pas de dialectes : les programmes C sont relativement portables

la concision des programmes nuit à la clarté

l'absence de vérification de type diminue la détection d'erreurs à la compilation

le programmeur suit souvent des conventions non imposées par le langage

la sémantique de certains opérateurs et types n'est pas définie précisément

la sémantique est ardue pour ceux qui n'ont pas une expérience de l'assembleur

1.3. Bibliographie

Le langage C, 2e édition
B. Kernighan & D. Ritchie
Masson - 1990

première référence du C par le père du langage, réédité pour le C ANSI

Langage C - Manuel de référence
S.P. Harbison & G.L. Steele Jr.
Masson - 1990

référence technique très pointue, indispensable à ceux qui veulent réaliser un compilateur C

Langage C - Les finesses d'un langage redoutable
J. Charbonnel
Armand Colin - Collection U - 1992

guide pédagogique, support de ce cours, exposant également les principaux pièges du langage

1.4. Les mots clés

symbole défini et réservé par le langage

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

1.5. Les constantes

constante entière

```
1   112   -71                                /* décimales */

0x12   0xFFFF   0X1A9F   0x1a9f          /* hexadécimales */

012   -077   01234                          /* octales */
```

constante réelle

```
123.   97.54   0.1   2e6    -77e-7
```


constante caractère

```
'a'   'A'   '0'

'\n'   '\077'   '\x1B'
```

caractère	octal	hexa	fonction	nom
\0	\000	\x00	caractère nul	NUL
\a	\007	\x07	signal sonore	BELL
\b	\010	\x08	retour arrière	Backspace
\f	\014	\x0C	saut de page	Form Feed
\n	\012	\x0A	saut de ligne	Line Feed
\r	\015	\x0D	retour chariot	Carriage Return
\t	\011	\x09	tabulation	Tab

caractère	hexa	nom
\c	63	c
\\	\x5C	backslash
\'	\x27	quote
\"	\x22	guillemet

constante chaîne de caractères

chaîne	décomposition en caractères
"Hello"	'H' 'e' 'l' 'l' 'o' '\0'
" "	'\x20' '\0'
""	'\0'
"\""	'"' '\0'
"0"	'0' '\0'
"\0"	'\0' '\0'

pas de limite à la longueur d'une chaîne

```
"ceci est une chaîne très très très très très \
très très très très longue..."

"ceci est une chaîne très très très très très "
  "très très très très longue..."
```


1.6. Opérateurs

```
+     -     *     /     %     ^     &     |     ~
!     =     <     >     +=    -=    *=    /=    %=
^=    &=    |=    <<    >>    <<=   >>=   ==    !=
<=    >=    &&    ||    ++    --    .     ,     ?:
->    ()    []    sizeof
```

1.7. Séparateurs

un séparateur est

- un signe de ponctuation (blanc, tabulation, saut de ligne)
- un commentaire

```
/* ceci est un commentaire assez long qui s'étend sur
plusieurs lignes */

/* ceci est un commentaire incorrect puisqu'il
contient /* un commentaire */ imbriqué */
```

1.8. Identificateurs

identifie variable, fonction, constante, type...

composé de lettres, de chiffres et de '_', le 1er caractère n'étant pas un chiffre

minuscules et majuscules significatives

ne doit pas être un mot clé

pas de limite de longueur, mais seuls les premiers caractères sont significatifs

```
nbcol   NbCol   NBCOL   nb_col
```

déclaration

introduction d'un identificateur dans le programme

précise le type de l'identificateur

définition

un identificateur doit être défini avant d'être utilisé

le compilateur implante l'objet en mémoire

éventuellement suivie d'une initialisation

exemple d'identificateur : les noms des variables

```
extern int x ;        /* déclare une variable entière x */

int x ;                 /* définit une variable entière */

int x=1 ;      /* définit un entier et l'initialise à 1 */

int a=2, b, c=-1 ;      /* définit 3 variables entières */
```

autre exemple d'identificateur : les noms des fonctions

une fonction est définie par son en-tête et son corps

```
ecrire()
{
  puts("bonjour !") ;
}
```

une fonction se déclare par son en-tête suivi d'un point-virgule

la déclaration d'une fonction est appelée également prototype

```
ecrire() ;            /* prototype de la fonction ecrire */
```

une fonction est appelée par son nom :

```
ecrire() ;                      /* appel de la fonction */
```

1.9. Structure d'un programme

succession de déclarations et de définitions de fonctions, variables et types

des commentaires et des directives pour le préprocesseur peuvent parsemer le programme

une des fonctions doit s'appeler `main`

l'exécution du programme = exécution de la fonction `main`

```
/* Simple programme C */
/* ------------------ */

#include <stdio.h>              /* pour le préprocesseur */

int carre(int) ;       /* prototype de la fonction carre */

main()                 /* définition de la fonction main */
{
  int i,j ;                 /* déclaration des variables */

  scanf("%d",&i) ;
  j = carre(i) ;
  printf("%d\n",j) ;
  return 0 ;
}

int carre(int i)      /* définition de la fonction carre */
{
  return i*i ;
}
```


1.10. La bibliothèque standard

le langage ne dispose pas d'instructions d'entrées/sorties, de gestion de la mémoire, de gestion de fichiers, de calcul mathématique...

on doit faire appel à des fonctions de la bibliothèque standard C

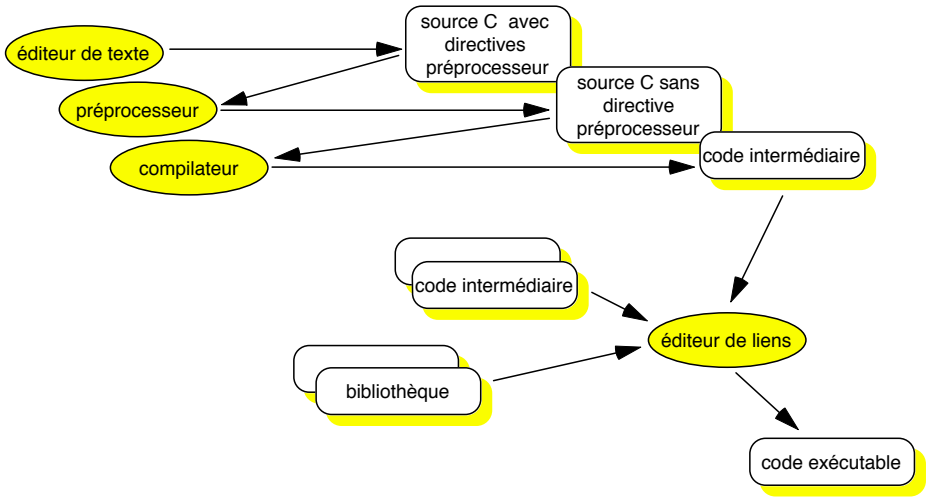
- **fonction putchar{XE "putchar"}**

```
putchar ('C') ;                /* écrit C sur la console */
```

- **fonction puts{XE "puts"}**

```
puts ("ceci est une chaîne de caractères") ;
                        /* écrit la chaîne sur la console */
```

1.11. Transformation d'un programme source en exécutable



1.12. Exercice

exercice 1

Réaliser un programme qui affiche *ceci est mon premier programme C* sur l'écran.

2. Les types de base

le type d'un objet précise :

- le mode de représentation
- la taille mémoire nécessaire à son stockage
- les opérations qui peuvent lui être appliquées

types de base et les types dérivés

les types de base correspondent aux unités élémentaires de stockage de l'ordinateur

2.1. Les types char, signed char et unsigned char

`char` est utilisé pour représenter les caractères de l'ordinateur

sa taille est 1 octet

sa valeur est un nombre correspondant au code du caractère dans le jeu de caractères de la machine

les tailles des autres types sont exprimées en terme de multiples de la taille d'un `char`

`sizeof(char)` vaut 1

`signed char` est toujours signé (plage de variation : 0..255)

`unsigned char` est toujours non signé (plage de variation : -128..127)

```
char c='A', cc, car=0x02 ;
char nl='\n' ;
unsigned char octet=0x02 ;
```


2.2. Les types int, unsigned, short et long

`int` convient à l'arithmétique entière

signé

sa taille dépend du processeur employé

```
int cpt, nbMots=0, i=-123 ;
```

`int` est le type implicite en C (compris entre `INT_MIN` et `INT_MAX`)

`unsigned` est synonyme de `unsigned int` (plage 0..`UINT_MAX`)

`short` et `long` sont également des types entiers

`short` est synonyme de `short int` (plage : `SHRT_MIN`..`SHRT_MAX`)

`long` est synonyme de `long int` (plage de variation : `LONG_MIN`..`LONG_MAX`)

```
static x ;                              /* x est entier */
const y=11 ;                            /* y est entier */
```

`unsigned int` est non signé

`unsigned short` (plage de variation : 0..`USHRT_MAX`)

`unsigned long` (plage de variation : 0..`ULONG_MAX`)

```
unsigned u ;
short int s1, s2 ;
unsigned long ul = 123456 ;
```

constantes

une constante entière dont la valeur dépasse le plus grand `int` est de type `long`

sinon la constante est de type `int`

une constante immédiatement suivie de la lettre `l` ou `L` est de type `long`

```
45000, 07L, 0xFFFF, 0xFFFl
```


2.3. Les types float et double

`float` représente des valeurs flottantes en simple précision

sa taille est généralement de 32 bits

le type `double` représente des valeurs à virgule flottante en double précision

sa taille est `2*sizeof(float)`.

```
float prix, taxe, total ;
double distance=254.5 ;
```

2.4. Récapitulatif

type	mot machine 32 bits	mot machine 64 bits
char	8 bits	8 bits
short	16 bits	16 bits
int	32 bits	32 bits
long	32 bits	64 bits
float	32 bits	32 bits
double	64 bits	64 bits

2.5. Le type void

`void` indique un ensemble vide de valeur

une fonction de type `void` ne renvoie pas de valeur

un pointeur sur un type `void` pointe sur une zone sans type

une liste de paramètres formels égale à `void` indique une liste vide

3. Les expressions

3.1. L'expression élémentaire

constante

variable

appel de fonction

```
3   x   f()
```

3.2. Lvalue

expression localisée en mémoire

une lvalue possède une adresse

une variable est une lvalue

`5` n'est pas une lvalue

3.3. L'expression composée

l'opérateur d'affectation

exp1=exp2 est une expression dont la valeur est *exp2*

la valeur de *exp2* est affectée à *exp1*

```
a=2    /* cette expression vaut 2, a reçoit la valeur 2 */

a=b=0                          /* équivalent à a = (b=0) */
```


les opérateurs arithmétiques

`+  -  *  /  %  ++  --`

exp1+exp2 , *exp1-exp2* , *exp1*exp2* , *exp1/exp2* et *exp1%exp2* sont des expressions dont la valeur est respectivement la somme, la différence, le produit, le quotient et le modulo des valeurs de *exp1* et *exp2*

\+ et – peuvent être utilisés comme opérateurs unaires

```
a3 = 2*a2 - a1
+x
-z
```

exp++ est une expression dont la valeur est *exp* et *exp* est incrémentée

++exp est une expression dont la valeur est *exp+1* et *exp* est incrémentée

exp-- est une expression dont la valeur est *exp* et *exp* est décrémentée

--exp est une expression dont la valeur est *exp-1* et *exp* est incrémentée

```
a++
--b
x + y++
3++
(2*z)++

int a, b=6 ;
a = ++b ;                          /* a==7 et b==7 */
a = b++ ;                          /* a==7 et b==8 */
++b ;                              /* a==7 et b==9 */
a = --b ;                          /* a==8 et b==8 */
a = b-- ;                          /* a==8 et b==7 */
```

les opérateurs de modification

`+=  -=  *=  /=  %=  ^=  |=  <<=  >>=`

exp1 op= exp2 où *op* est un opérateur a pour valeur *exp2*

exp1 op= exp2 est équivalente à *exp1=exp1 op exp2*

```
z *= 2                          /* équivalent à z = z*2 */

x += 1                /* équivalent à x = x + 1 et à x++ */
```

les opérateurs relationnels

`==  !=  <  >  <=  >=`

0 est considéré comme la valeur FAUX

tout autre entier est considéré comme la valeur VRAI

exp1==exp2 vaut 1 si *exp1* et *exp2* ont la même valeur, 0 sinon

exp1!=exp2 vaut 1 si *exp1* et *exp2* ont des valeurs différentes, 0 sinon

exp1<exp2 , *exp1>exp2* , *exp1<=exp2* et *exp1>=exp2* valent 1 si *exp1* a une valeur respectivement inférieure, supérieure, inférieure ou égale et supérieure ou égale à celle de *exp2*, 0 sinon

```
1!=2
x = y==z                      /* équivalent à x = (y==z) */
```


les opérateurs logiques

`||  &&  !`

exp1||exp2 vaut 0 si les deux expressions valent toutes deux 0, 1 sinon

exp1&&exp2 vaut 0 si l'une au moins des expressions vaut 0, 1 sinon

exp2 est évaluée seulement si c'est nécessaire

!exp vaut 1 si l'expression *exp* vaut 0, 0 sinon

```
1!=2 || 3>5

if ( x>=0 && sqrt(x)<epsilon ) ...

while (i<max && tab[i]!=alpha) ...
```

les opérateurs d'indirection

`&  *`

&exp donne l'adresse de l'expression, qui doit être une lvalue

**exp* est l'objet contenu à l'adresse *exp*

l'opérateur conditionnel

`? :`

exp1`?`*exp2*`:`*exp3* vaut *exp3* si *exp1* vaut 0, et la valeur de *exp2* sinon

```
abs = x>0 ? x : -x
min = x<y ? x : y
```

l'opérateur d'évaluation séquentielle

exp1`,`*exp2* vaut *exp2*, *exp1* est évaluée, sa valeur est perdue, puis *exp2* est à son tour évaluée

priorités et associativité des opérateurs

opérateurs classés par priorité décroissante :

opérateurs	fonction	associativité
-> . [] ()	sélection de champ indiçage appel de fonction	gauche
++ -- ~ ! + - & * () sizeof	pré- et post-incrémentation, pré- et post-décrémentation complément à 1 négation plus et moins unaire référence et indirection cast taille	droite
* / %	multiplication, division modulo	gauche
+ -	addition et soustraction	gauche
<< >>	décalage à gauche et à droite	gauche
< > <= >=	inférieur et supérieur inférieur ou égal et supérieur ou égal	gauche
== !=	égal et différent	gauche
&	ET bit à bit	gauche
^	OU exclusif bit à bit	gauche
|	OU inclusif bit à bit	gauche
&&	ET logique	gauche

`||`	OU logique	gauche
`?:`	conditionnel	droite

`=`
`+=  -=  *=  /=  %=`
`<<=  >>=  &=  |=  ^=`
	affectation
	modification
		droite

`,`	évaluation séquentielle	gauche

3.4. L'expression constante

expression constante = expression évaluable à la compilation

peut être composée :

- de constantes entières ou caractères (par exemple `5` ou `'C'`)
- d'opérateurs unaires (`- ~`)
- d'opérateurs arithmétiques (`+ - * / %`)
- d'opérateurs relationnels (`== != < > <= >=`)
- d'opérateurs logiques (`&& || !`)
- d'opérateurs de manipulation de bits (`& | ^ << >>`)
- de l'opérateur conditionnel (`? :`)
- des opérateurs `sizeof` et `defined`

elle peut contenir également des symboles et des macros

4. Les instructions

4.1. L'instruction élémentaire

instruction élémentaire = expression suivie par un point-virgule

la valeur de l'expression est perdue

```
x=1 ;          /* la valeur de l'affectation est perdue */
x=y=1 ;                    /* équivalent à x = (y=1) ; */
f(3) ;           /* la valeur de retour de f est perdue */
x + 1 ;                          /* instruction inutile */
g() ;  /* appel de la fonction g et abandon du résultat */
g ;                              /* instruction inutile */
```


4.2. L'instruction composée

bloc contenant des déclarations, définitions et instructions

entouré d'accolades

peut contenir d'autres blocs

4.3. Les déclarations dans un bloc

possible de déclarer des identificateurs en début, avant la première instruction

4.4. Les définitions dans un bloc

possible de définir des types, des variables

impossible de définir des fonctions

les définitions doivent précéder la première instruction

les entités définies en début de bloc sont locales à ce bloc

ne sont visibles que dans le bloc de définition et dans ses sous-blocs

masquent les variables de mêmes noms définies à l'extérieur du bloc

les variables locales sont automatiques : elles naissent à l'entrée du bloc et sont détruites à la sortie

```
int i=11 ;
...
{
  int i ;
  ...
  i=0 ;
  {
    int i=5 ;
    i++ ;                                   /* i vaut 6 */
    ...
  }
  i++ ;                                     /* i vaut 1 */
  ...
}
```


➢ fonction printf{XE "printf"}

```
int i=27 ;
char c='A' ;
char s[40]="chaîne" ;
float x=1.23 ;
long l=700000 ;
unsigned u=66000 ;

printf ("i=%d c=%c s=%s x=%f\n",i,c,s,x) ;
                        /* i=27 c=A s=chaîne x=1.2300000 */

printf ("%x %4d %-4d %04d\n",i,i,i,i) ;
                                    /* 1B   27 27   0027 */

printf ("%d %x %6.2f\n",c,c,x) ;        /* 65 41   1.23 */

printf ( "%u %ld\n" , u , l ) ;         /* 66000 700000 */
```

➢ fonction scanf{XE "scanf"}

```
int i ;
char c ;
char s[40] ;
float x ;

scanf ("%d",&i) ;           /* tente de lire un entier */
scanf ("%c",&c) ;         /* tente de lire un caractère */
scanf ("%s",s ;             /* tente de lire une chaîne */
scanf ("%f",&x) ;              /* tente de lire un réel */
```

4.5. Exercice

exercice 2

Ecrire un programme qui lit deux nombres au clavier et affiche le plus grand des deux.

exercice 3

Ecrire un programme qui demande un nombre entier positif et affiche son factoriel.

5. Les fonctions

5.1. En-tête

entête = type de retour + identificateur de fonction + (liste paramètres formels)

les parenthèses sont obligatoires même si la liste est vide

les paramètres sont séparés par une virgule

pour chaque paramètre doivent être précisés son type et son nom

```
int f(float x,float y)
void g(char c)
double h()
float reset(void)
```

5.2. Corps de fonction

corps de fonction = instruction composée

5.3. Valeur de retour

valeur renvoyée par la fonction via `return`

si le type de la fonction est `void`, la fonction ne renvoie pas de valeur

`return` renvoie l'exécution à la fonction appelante

si `return` est suivi d'une expression, celle-ci devient la valeur de la fonction et doit être compatible avec son type

un `return` sans expression peut être utilisé si la fonction est de type `void`

```
int carre(int i)
{
  int result ;

  result = i*i ;
  return result ;
}
```

sans `return`, une fonction se termine après exécution de la dernière instruction

5.4. Appel de fonction

appel de fonction = nom suivi de la liste des paramètres effectifs entre parenthèses

les parenthèses sont obligatoires, même s'il n'y a aucun paramètre

```
int carre(int i)
{
  return i*i ;
}

main()
{
  int k, t ;
  scanf ( "%d" , &k) ;
  t = carre(k) ;
  printf ( "%d", t ) ;
}
```

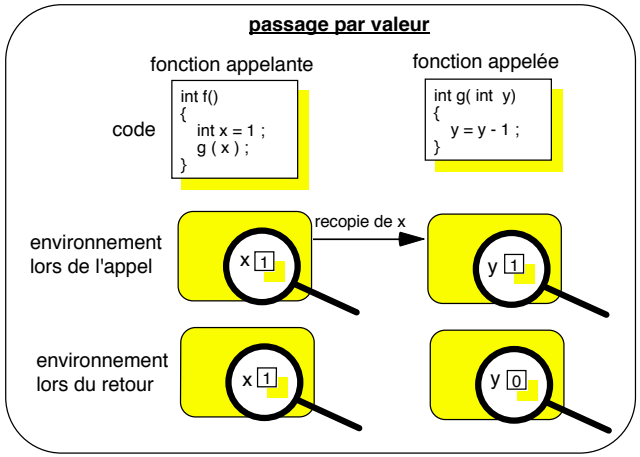
5.5. Passage de paramètres

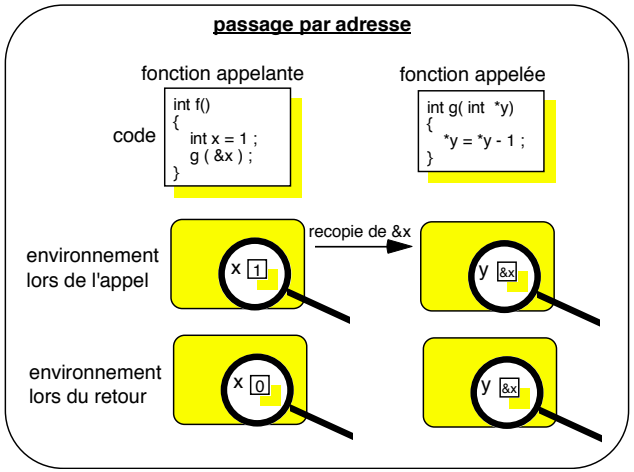
les paramètres des fonctions sont passés exclusivement par valeur

chaque paramètre formel est initialisé avec la valeur du paramètre effectif de même rang

le paramètre formel est la copie du paramètre effectif de même rang

pour qu'une fonction appelée modifie une variable de la fonction appelante, il faut passer son adresse





5.6. Prototype

prototype de la fonction = déclaration de la fonction

en-tête de la fonction dans lequel le nom des paramètres formels est facultatif

```
int carre(int) ;
```

une liste vide de paramètres signifie que l'on ne précise pas les paramètres

pour indiquer l'absence de paramètre, il faut mettre `void`

```
float f() ;
float f(void) ;
```


5.7. L'ellipse

"`...`" permet de ne fixer que les premiers paramètres de la liste

permet la réalisation de fonctions capables de traiter un nombre variable d'arguments de type variable

```
int moyenne(int nb, ...)
{
    ...
}
```

5.8. La syntaxe K&R

```
float f(int a, char c)                    /* syntaxe ANSI */
{
    return a / c ;
}

float f(a,c)                  /* syntaxe K&R équivalente */
int a ;
char c ;
{
    return a / c ;
}
```

5.9. Exemple

```
        /* tri de 3 entiers  lus sur l'entrée standard */
#include <stdio.h>
                    /* renvoie le plus grand de a et b */
int max(int a, int b)
{ return a>b ? a : b ; }
                    /* renvoie le plus petit de a et b */
int min(int a, int b)
{ return a<b ? a : b ; }
                  /* renvoie le plus grand de a, b et c */
int premier(int a, int b, int c)
{ return max ( a, max(b,c) ) ; }
                  /* renvoie le plus petit de a, b et c */
int troisieme(int a, int b, int c)
{ return min ( a, min(b,c) ) ; }

int deuxieme(int a, int b, int c)
{ return troisieme ( max(a,b) , max(b,c) , max(a,c) ) ; }

void main()
{
  int x, y, z ;
  printf("entrer 3 entiers : ") ;
  scanf("%d %d %d", &x, &y, &z ) ;
  printf("les voici en ordre croissant : %d %d %d\n",
   troisieme(x,y,z), deuxieme(x,y,z), premier(x,y,z) ) ;
}
```


6. Le préprocesseur

prépare la source en C pour le compilateur

programme indépendant ou intégré au compilateur

en fonction des directives qu'il rencontre, le préprocesseur traite la source C et génère un texte C intermédiaire

les directives sont éliminées après leur traitement

la source intermédiaire est toujours du C et est ensuite compilée

directive = ligne commençant par #

si le dernier caractère de la ligne est \, la directive continue sur la ligne suivante

une directive peut apparaître n'importe où dans le programme source

6.1. Définir des symboles

#define symbole chaîne_de_remplacement

simple substitution de chaînes de caractères ne faisant pas intervenir les règles syntaxiques du langage C

```
#define PI 3.14159265358979323846264338327950288419716939937
#define PIPI 2*PI

#define BEGIN {
#define END }

#define BIP_BIP BEGIN putchar('\a') ;  putchar('\a') ; END

#define BANNIERE "Compilateur C - version 3.01 - 1992"

void main()
BEGIN
  puts ( BANNIERE ) ;
  ...
  if ( ... ) BIP_BIP ;
  ...
END

const double pi = 3.1415926535898 ;
```


6.2. Supprimer des symboles

#undef *symbole*

```
#undef PI
#define PI (355/113)
```

6.3. Inclure des fichiers

#include <nom_de_fichier**>**

#include "nom_de_fichier**"**

les directives `#include` peuvent être imbriqués.

6.4. Les fichiers en-têtes

`stdlib.h`	traitements courants
`stdio.h`	entrées/sorties
`string.h`	manipulation de chaînes de caractères
`ctype.h`	classification et conversion de caractères
`math.h`	traitements mathématiques
`time.h`	définition des types et des fonctions de manipulation du temps
`limits.h`	définition des caractéristiques de l'implémentation (tailles des types, valeurs extrêmes...)

6.5. Exercice

exercice 4

Compléter le programme factoriel de façon à supprimer tous les avertissements du compilateur.

7. Les instructions de contrôle

7.1. Les instructions conditionnelles

if (*expression* **)** *instruction*

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

```
if (corps==REEL || corps==COMPLEX)
  if ( delta>0 )
    puts("2 solutions réelles") ;
  else if (delta==0)
             puts("1 solution réelle double") ;
          else if (corps==COMPLEX)
                     puts("2 solutions imaginaires") ;
                  else
                     puts("pas de solution") ;
else
  {
    puts("résolution non prévue") ;
    exit(1) ;
  }
```


7.2. L'instruction sélective

switch (*expression* **)**
{
case *expression_constante_1* **:** *instructions_1*
case *expression_constante_2* **:** *instructions_2*
.....
}

switch (*expression* **)**
{
case *expression_constante_1* **:** *instructions_1*
case *expression_constante_2* **:** *instructions_2*
.....
default : *instructions_n*
}

```
switch ( piece )
{
 case 0: puts("pion noir");
         break ;
 case 1: puts("dame noire");
         break ;
 case 2: puts("pion blanc");
         break ;
 case 3: puts("dame blanche");
         break ;
 default: puts("cas non prévu") ;
          exit(1) ;
}
```

```
switch ( x )
{
  case 'o' :
  case 'O' :
    ok=1 ;
    break ;

  case 'n' :
  case 'N' :
    ok=0 ;
    break ;

  default :
    ok=-1 ;
}
```

7.3. Les instructions itératives

while

while (*expression*) *instruction*

```
scanf("%d",&n) ;
while ( n<0 || n>100 )
{
  puts("hors borne") ;
  scanf("%d",&n) ;
}
```


do while

do *instruction* **while** (*expression*)

```
do {
     scanf("%d",&n) ;
     if ( n<0 || n>100 ) puts("hors borne") ;
   }
while ( n<0 || n>100 ) ;
```

for

for (*exp1* ; *exp2* ; *exp3*) *instruction*

```
exp1
while ( exp2 )
{
  instruction
  exp3
}
```

```
for (i=0 ; i<NB ; i++ ) printf("%d ",i) ;

for ( i=1, j=0 ; i!=0 && i>j ; i*=2, j++ ) { ... }

for (;;) { ... }
```

7.4. Les instructions de rupture de séquence

break

`break` provoque la sortie d'une instruction `switch`, `while`, `do while` ou `for`

```
for (;;)
{
  ...
  if ( ... ) break ;
  ...
  if ( ... ) break ;
  ...
  if ( ... ) break ;
  ...
}
```


7.5. Exercice

exercice 5

Ecrire un programme calculant $\sum_{i=1}^{\infty} \frac{1}{i^2}$

8. Les pointeurs

T **identificateur* ;

si *T* est un type, *T** est le type *pointeur sur T*

une variable de type *T** peut contenir l'adresse d'une zone mémoire susceptible de contenir elle-même un élément de type *T*

la taille d'une variable de type *T** est indépendante de *T*

```
int* ptr ;

int *ptr ;

int* a, b ;

int *f() ;

int (*f)() ;
```

```
int i, *ptr ;
ptr = &i ;          /* initialise ptr à l'adresse de i */
*ptr = 11 ;       /* initialise la zone pointée par ptr */
```

le type `void *` est un pointeur sans type

il n'est pas possible d'effectuer une indirection sur un pointeur de type `void`

```
void *ptr ;
*ptr = 1 ;                                /* incorrect */
```

`NULL` : constante pointeur qui ne pointe sur rien

```
if (ptr!=NULL) i=*ptr ;
```


remarques

le nom d'une fonction est interprété comme étant l'adresse de cette fonction

le nom d'un tableau est interprété comme étant l'adresse du premier élément de ce tableau

utiliser `const` pour définir un pointeur constant :

```
char tab[20] ;
...
char * const ptr = tab ;/* le pointeur ptr est constant */
ptr = &tab[3] ;                              /* illégal */
ptr[2] = 'a' ;                                  /* ok */
```

utiliser également `const` pour définir un pointeur vers une constante :

```
const char *ptr = tab ;   /* tab non modifiable via ptr */
ptr = &tab[3] ;                                 /* ok */
ptr[2] = 'a' ;                               /* illégal */
```

8.1. L'arithmétique des pointeurs

addition et soustraction avec un entier

si `p` est de type *T**, et `i` un entier,

- `p+i` = l'adresse du i[ème] élément de type *T* situé après l'adresse `p`
- `p-i` est l'adresse du i[ème] élément de type *T* situé avant `p`

opérations impossibles avec les pointeurs de type `void *`

soustraction de 2 pointeurs

Si `p` et `q` sont deux pointeurs de même type, `p-q` est le nombre d'éléments qui séparent `*q` de `*p`

opérations impossibles avec les pointeurs de type `void *`

relations avec les pointeurs

Il est possible de comparer les valeurs de deux pointeurs avec les opérateurs relationnels standard (`== != < > <= >=`)

initialisation

```
int *ptr ;
*ptr = 1 ; /* range la valeur 1 à une adresse indéfinie */

int i, *adr=&i, *ptr ;
ptr = &i-1 ;

int *ptr = adr ;
```

certaines fonctions permettent d'initialiser un pointeur (fonctions d'allocation de mémoire)

8.2. La gestion dynamique de la mémoire

➢ fonction malloc{XE "malloc"}

```
int *adr_zone ;
adr_zone = malloc(100*sizeof(int)) ;      /* allocation */
if ( adr_zone==NULL )
  { puts("mémoire insuffisante") ; exit(1) ; }
```

➢ fonction free{XE "free"}

il n'existe pas en C de ramasse-miettes

```
free(adr_zone) ;                          /* libération */
```


8.3. Exercices

exercice 6

Ecrire une fonction `swap`, à deux paramètres, qui permute les valeurs de 2 variables entières quelconques.

exercice 7

Connaissant la somme et le produit de 2 entiers `a` et `b`, écrire une fonction qui admet en paramètre d'entrée la somme et le produit et renvoie en paramètre de sortie `a` et `b`.

9. Les tableaux

9.1. La définition

si *T* est un type, on peut construire un objet de type *tableau de T*

T identificateur **[** *expression_constante* **]**

chaque élément du tableau est de type *T*

```
int tab[10] ;

int damier[8][8] ;
float cube[10][10][10] ;
```

9.2. Les éléments

indicés de 0 au nombre d'éléments -1

```
int tab[5] ;       /* définit tab[0], tab[1]..., tab[4] */
```

```
damier[1][5]
cube[2][3][1]
```

```
float mat[2][2] ;
     /* définit mat[0][0] mat[0][1] mat[1][0] mat[1][1] */
```


9.3. L'initialisation

```
int  tab[10] = { 1,0,3,2,5,4,7,6,9,8 } ;
char msg[6] = { 'e','r','r','e','u','r' } ;

int  tab[] = { 1,0,3,2,5,4,7,6,9,8 } ;
char msg[] = { 'e','r','r','e','u','r' } ;



int matrice[4][2] = { 1,2,2,3,4,4,5,6 } ;

int matrice[4][2] = {                          { 1, 2 },
                                               { 2, 3 },
                                               { 4, 4 },
                                             { 5, 6 } } ;



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

int tab[5] = { 1, 2, 3, 0, 0 } ;



float matrice[3][3] = { { 1, 2 }, { 3 } } ;

float matrice[3][3] = {                      { 1, 2, 0 },
                                           { 3, 0, 0 } ,
                                         { 0, 0, 0 } } ;
```

tableaux et pointeurs

dans toute expression, `tab` est converti en un pointeur sur son premier élément :

- `tab` est converti en `&tab[0]`
- `tab[i]` est équivalent à `*(tab+i)`
- `tab[1][2]` est équivalent à `*(*(tab+1)+2)`

```
char *ptr ;   /* alors ptr[i] est équivalent à *(ptr+i) */
```

```
int *tab ;
tab = malloc(nb*sizeof(int)) ;
for ( i=0 ; i < nb ; i++ )
  scanf("%d",&tab[i]) ;
...
free(tab) ;
```


9.4. Tableaux et fonctions

`f(tab)` est convertie en `f(&tab[0])`

par conséquent le tableau se retrouve transmis par adresse

toute modification de ses éléments se répercute dans la fonction appelante

un tableau peut être passé en paramètre à une fonction qui attend un pointeur, et réciproquement

si `tab` et `ptr` sont déclarés par :

```
int tab[10] ;
int *ptr ;
```

les appels `f(tab)` et `f(ptr)` sont valides pour toute fonction `f` déclarée comme :

```
void f(int [10]) ;
void f(int []) ;
void f(int *) ;
```

9.5. Opérations sur les tableaux

pas d'affectation entre tableaux

pas de comparaison de deux tableaux

`tab1=tab2` provoque une erreur de compilation

➢ fonction memcpy{XE "memcpy"}

```
memcpy(tab1,tab2,sizeof(tab2)) ;    /* recopie de zones */
```

`tab1==tab2` est valide, mais teste si les deux tableaux commencent à la même adresse

➢ fonction memcmp{XE "memcmp"}

```
memcmp(tab1, tab2, max(sizeof(tab1),sizeof(tab2))) ;
                                /* comparaison de zones */
```


9.6. Exercices

exercice 8

L'algorithme du tri à bulles permet de trier les éléments d'un tableau. Il consiste à parcourir le tableau et à permuter toute paire d'éléments consécutifs qui ne sont pas dans le bon ordre. Le processus est réitéré jusqu'à ce qu'un parcours complet se soit réalisé sans qu'aucune permutation n'ait eu lieu.

Ecrire un programme de tri, en utilisant par exemple l'algorithme ci-dessus : les objets à trier sont des entiers, éléments d'un tableau dimensionné statiquement. On supposera que le nombre d'entiers est `N`, défini en constante dans le programme. On pourra comparer une version *naturelle* utilisant des indices pour parcourir le tableau avec une autre n'utilisant que des pointeurs.

exercice 9

Ecrire un programme de tri : les objets à trier sont des entiers stockés dans une zone de mémoire allouée dynamiquement. Le nombre de valeurs à trier est saisi au clavier et une zone d'une taille juste nécessaire à leur stockage est créée dynamiquement.

10. Les chaînes de caractères

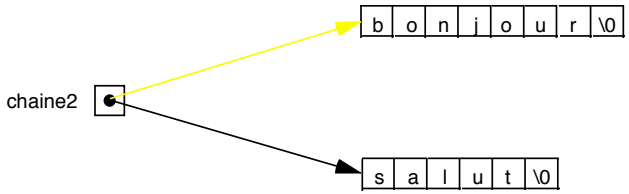
10.1. Généralités

chaîne de caractères = tableau de caractères

le dernier caractère significatif est le caractère '\0'

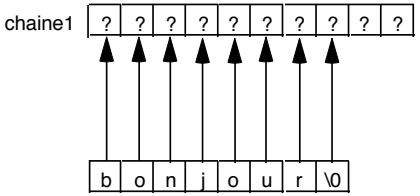
différencier la longueur maximale de la chaîne de la longueur réelle

```
char chaine[10]={'b','o','n','j','o','u','r','\0'};

char chaine[10]="bonjour";       /*le '\0' est implicite*/

char chaine1[] = "bonjour" ;

char *chaine2 = "bonjour"
                  /* pointeur initialisé sur "bonjour" */


chaine2="salut";
```



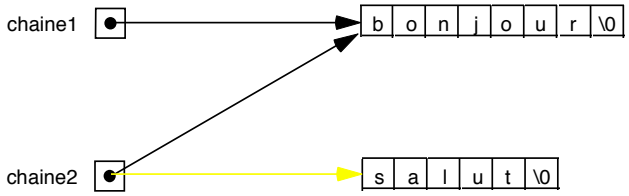
➢ **fonction strcpy{XE "strcpy"}**

```
char chaine1[10];
chaine1="bonjour";                          /* illégal */
strcpy(chaine1,"bonjour") ;                   /* ok */
```

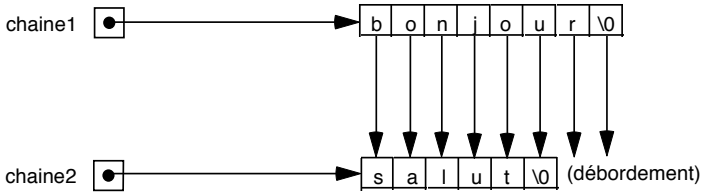


```
char *chaine1="bonjour", *chaine2="salut" ;
```

```
chaine2 = chaine1 ;
```




```
strcpy(chaine2,chaine1) ;
```



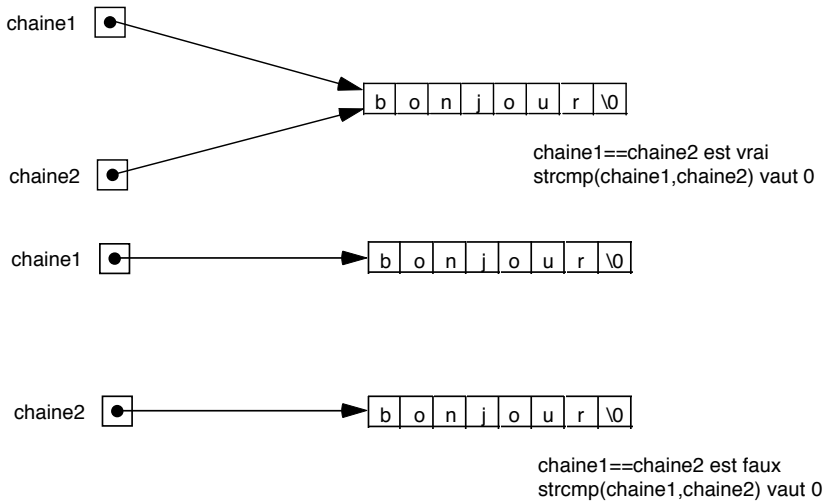
les fonctions de la bibliothèque standard supposent que toute chaîne se termine par `'\0'`

```
char chaine1[6]="salut", chaine2[6] ;
chaine1[5] = '!' ; /* chaine1 n'est plus terminée par \0*/
strcpy(chaine2,chaine1) ;
```


➢ fonction strcmp{XE "strcmp"}

```
char *chaine1="bonjour", *chaine2="bonjour"
chaine1==chaine2                /* compare les adresses */
```

```
strcmp(chaine1,chaine2) ;       /* compare les contenus */
```



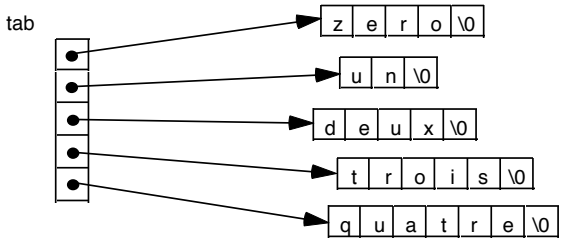
10.2. Tableaux de chaînes de caractères

tableaux de tableaux de caractères

```
char tab[5][10] = { "zero", "un", "deux",
                    "trois", "quatre" } ;
```

```
z  e  r  o  \0 ?  ?  ?  ?  ?
u  n  \0 ?  ?  ?  ?  ?  ?  ?
d  e  u  x  \0 ?  ?  ?  ?  ?
t  r  o  i  s  \0 ?  ?  ?  ?
q  u  a  t  r  e  \0 ?  ?  ?
```

```
char *tab[5] = { "zero", "un", "deux",
                 "trois", "quatre" } ;
```




```
char **tab ;
```

- une première allocation nécessite de connaître le nombre de chaînes
- pour chaque chaîne doit être faite une allocation du nombre de caractères{XE "strlen"}

```
char **tab, buffer[MAX] ;
tab = malloc(nb*sizeof(char *)) ;
for ( i=0 ; i<nb ; i++ )
{
  gets(buffer) ;
  lg = strlen(buffer) ;        /* strlen renvoie le nombre
                                de caractères d'une chaîne */
  tab[i] = malloc((lg+1)*sizeof(char)) ;
  strcpy(tab[i],buffer) ;
}
```

10.3. Exercices

exercice 10

Ecrire une fonction `len` qui admet un paramètre de type chaîne de caractères et retourne le nombre de caractères de cette chaîne.

exercice 11

Ecrire une fonction `cpy` qui admet deux paramètres de type chaîne de caractères et qui recopie tous les caractères de la deuxième chaîne dans la première.

N.B. : la fonction `cpy` supposera que la taille de la chaîne réceptrice est suffisamment grande.

exercice 12

Ecrire une fonction qui admet en paramètre une chaîne de caractères et qui renvoie en retour VRAI ou FAUX suivant que la chaîne est ou non un palindrome.

exercice 13

Ecrire une fonction qui admet en paramètre deux chaînes de caractères et qui renvoie en retour VRAI ou FAUX suivant que les chaînes sont ou non anagramme.

11. La génération de types

les déclarateurs peuvent être composés

```
int *tab[10] ;   /* tableau de 10 pointeurs vers entier */
```

déclarateurs de tableau et de fonction prioritaires par rapport aux pointeurs

```
int *fct() ;              /* équivalent à int *(fct()) ; */

int (*fct)() ;

int *(*x)[10] ;
```

`*(*x)[i]` : `int` (entier)
`(*x)[i]` : `int *` (pointeur vers entier)
`*x` : `int *[10]` (tableau de 10 pointeurs vers `int`)
`x` : `int *(*)[10]` (pointeur vers tableau de 10 pointeurs vers `int`)

```
int *(*P)() ;
```

P est un pointeur vers une fonction, laquelle retourne un pointeur vers un entier

```
int (*T[])() ;
```

T est un tableau de pointeurs vers des fonctions renvoyant un entier

```
int *(*T[])() ;
```

T est un tableau de pointeurs vers des fonctions retournant un pointeur vers un entier

```
int (*F)() = f ;
```

F est un pointeur vers une fonction renvoyant un entier, ce pointeur est initialisé à l'adresse de la fonction f

```
long **F() ;
```

F est une fonction qui retourne un pointeur vers un pointeur vers un type long

```
short (*F())[] ;
```

F est une fonction renvoyant un pointeur vers un tableau d'éléments de type short

```
double *(*F())() ;
```

F est une fonction qui retourne un pointeur vers une fonction qui retourne un pointeur vers un type double

```
char (*(*(*F())[])())[] ;
```

F est une fonction qui retourne un pointeur vers un tableau de pointeurs vers des fonctions renvoyant un pointeur vers des tableaux d'éléments de type char

11.1. Typedef

```
typedef unsigned char Bool ;
typedef char String[255] ;
typedef int *Ptr ;

Bool fin, erreur ;
Ptr ptr, f(int) ;
String s, tab[10] ;
int t = sizeof(String) ;
```

11.2. Les types anonymes

déclarateur de type anonyme = déclarateur sans l'identificateur

`float *`	pointeur vers un réel
`char (*)()`	pointeur vers une fonction retournant un caractère
`unsigned *[10]`	tableau de 10 pointeurs vers un entier non signé
`int (*(*)())()`	pointeur vers une fonction retournant un pointeur vers une fonction renvoyant un entier

```
ptr = (char [255]) malloc(10*sizeof(char [255]))

double zero(double, double, double (*)(double)) ;

z = zero (1.57,4.71,sin) ;
```

11.3. Les conversions explicites

cast

```
int a=1, b=2 ;
float x ;

x = a/b                    /* division entière, x==0.0 */
x = (float) a/b ;      /* division réelle 1.0/2, x==0.5 */
x = a/(float)b ;       /* division réelle 1/2.0, x==0.5 */
x = (float) (a/b)          /* division entière, x==0.0 */
```


12. La fonction main

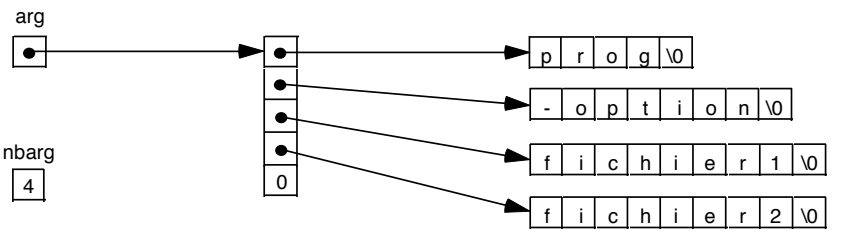
paramétrer un programme par des arguments fournis lors de son activation :

```
int main(int nbarg, char **arg)/* pour le programme prog */
```

les arguments de la commande sont récupérés via les paramètres de `main`

- un entier : le nombre d'arguments fournis
- un tableau de chaînes de caractères : les arguments eux-mêmes (le nom du programme est considéré comme le premier argument)

```
prog -option fichier1 fichier2
```



un programme peut renvoyer un code de fin d'exécution au processus père

c'est la valeur renvoyée par la fonction `main` ou par toute fonction de terminaison d'exécution, comme `exit`

par convention, un code nul indique que l'exécution s'est bien passée

```
int main(int nbarg, char *arg[])
{
  switch ( nbarg )
  {
    case 1 :   erreur() ;
               exit(1) ;
    case 2 :   traiter("",arg[1]) ;
               break ;
    case 3 :   if ( arg[1][0]!='-' )
               {
                 erreur() ;
                 exit(2) ;
               }
               else
                 traiter(arg[1],arg[2]) ;
               break ;
    default :  erreur() ;
               exit(3) ;
  }
  return 0 ;
}
```


Exercices

exercice 14

Ecrire un programme `say` qui écrit, sur la sortie standard, les arguments qu'il a reçu sur sa ligne de commande.

exercice 15

Modifier le programme `say` précédent de façon à interpréter quelques caractères spéciaux. Les caractères spéciaux sont composés d'un backslash suivi d'une lettre. On considèrera les séquences spéciales et leur interprétation suivantes :

- — `\n` saut de ligne
- — `\b` retour arrière
- — `\t` tabulation
- — `\\` backslash

Tout autre caractère précédé d'un backslash sera écrit tel quel, sans le backslash.

N.B. : les caractères spéciaux peuvent se trouver à l'intérieur des arguments.

13. Les énumérations

énumération = ensemble de constantes nommées

```
enum jour { dimanche, lundi, mardi, mercredi,
            jeudi, vendredi, samedi } ;
enum jour aujourdhui, demain ;

enum jour { dimanche, lundi, mardi, mercredi,
            jeudi, vendredi, samedi }
          aujourdhui, demain ;

enum { dimanche, lundi, mardi, mercredi, jeudi,
       vendredi, samedi } aujourdhui, demain ;

aujourdhui = lundi ;
demain = mardi ;
```

un entier est associée séquentiellement à chaque constante, à partir de 0

les opérations applicables aux entiers sont applicables aux énumérations

```
demain = aujourdhui + 1 ;

enum piece { fou=2, pion=1, cavalier, tour ,
             reine, roi=100 } ;
```

`enum piece` est le nom du nouveau type :

```
enum piece *blanc ;
blanc = (enum piece *) malloc(16*sizeof(enum piece)) ;
```

l'utilisation de `typedef` permet d'alléger les notations :

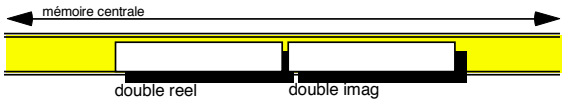
```
enum bool { FALSE, TRUE } ;
typedef enum bool Bool ;
Bool fini ;
```

14. Les structures

structure = type composé d'éléments de type quelconque appelés champs

14.1. La définition

```
struct complexe { double reel,imag ; } ;
struct complexe x,y ;
```



struct complexe x

```
struct complexe { double reel,imag ; } x,y ;

struct { double reel,imag ; } x,y ;

struct Date { short jour,mois,an ; } ;

struct Temps { short heure,minute,seconde ; } ;

struct Instant { struct Date date ;
                 struct Temps temps ; } ;
```


une référence à une structure peut être faite avant sa définition, à condition que sa taille ne soit pas nécessaire au compilateur :

```
struct chaine { int maillon ;
                struct chaine *suivant ; } ;
```

une structure peut être initialisée de la même façon qu'un tableau :

```
struct Instant maintenant = { 1,7,1990,9,30,0 } ;

struct Instant maintenant = { { 1, 7, 1990 } ,
                              { 9, 30, 0 } } ;
```

un nombre de constantes supérieur au nombre de champs de la structure provoque une erreur de compilation

pour un nombre inférieur, les éléments restants sont initialisés à 0

14.2. Les opérations sur les structures

l'opérateur . permet de référencer un champ dans une structure :

```
struct Date hier ;
hier.jour = 30 ;
hier.mois = 6 ;
hier.an = 1990 ;

struct Instant moment[10] ;
moment[0].date.jour = 1 ;
moment[0].date.mois = 7 ;
moment[0].date.an = 1990 ;
moment[0].temps.heure = 9 ;
moment[0].temps.minute = 35 ;
moment[0].temps.seconde = 0 ;
```

l'opérateur -> permet de référencer un champ à partir d'un pointeur de structure :

```
struct complex *z ;
z = malloc(sizeof(struct complex)) ;
z->reel = 0.1 ;
z->imag = -.033 ;
```

ce qui est équivalent à :

```
(*z).reel = 0.1 ;
(*z).imag = -.033 ;
```

l'opérateur = permet d'affecter une structure à une autre structure de même type

impossible de comparer globalement deux structures avec l'opérateur ==

14.3. Structures et fonctions

une structure peut être passée en paramètre à une fonction

elle est recopiée localement à l'entrée de la fonction :

```
struct S { int tab[20] ; } s ;
f(s) ;
```

une fonction peut retourner une structure :

```
typedef struct complex COMPLEX ;

COMPLEX init(double r, double i)
{
  COMPLEX nouveau ;
  nouveau.reel = r ;
  nouveau.imag = i ;
  return nouveau ;
}

COMPLEX add(COMPLEX a, COMPLEX b)
{
  COMPLEX somme ;
  somme.reel = a.reel + b.reel ;
  somme.imag = a.imag + b.imag ;
  return somme ;
}
```

14.4. Les champs de bits

champs de taille inférieure à celle d'un `char`

```
struct S { unsigned a : 1 ,
                    b : 3 ,
                    c : 5 ;
           int      d ;
         } ;
```

définit `S` comme ayant :

- un champ `a` d'un bit
- un champ `b` de 3 bits
- un champ `c` de 5 bits
- un champ `d` entier

les champs de bits sont toujours de type `unsigned`

```
struct S { unsigned a : 1 ,
                      : 3 ,
                    c : 5 ;
         } ;
```

laisse un trou de 3 bits entre les champs `a` et `c`

14.5. Exercice

exercice 16

Ecrire un programme d'évaluation d'une fonction mathématique quelconque de la bibliothèque standard (on se limitera aux fonctions ayant pour argument et pour valeur de retour un `double`, par exemple les fonctions trigonométriques `sin`, `cos`, `tan`, les fonctions inverses `asin`, `acos`, `atan`, les fonctions exponentielle et logarithmiques `exp`, `log`, `log10`, la fonction racine carrée `sqrt`, les fonctions hyperboliques `sinh`, `cosh`, `tanh`, etc.).

Le programme lit sur la ligne de commande le nom de la fonction et la valeur de son argument.

Le programme doit être conçu de façon à être facilement enrichissable par de nouvelles fonctions.

15. Les unions

type union = composé de plusieurs champs superposés en mémoire

une union ne peut contenir qu'une seule valeur à un instant donné

définition, initialisation et utilisation similaire à celle de la structure

les champs de bits n'y sont pas autorisés

la taille d'une union est la taille du plus large champ

```
union mois { int num ;
             char nom[10] ; } inventaire ;

inventaire.num = 2 ;

strcpy(inventaire.nom , "février" ) ;
```

union mois inventaire

impossible de savoir quel composant a été modifié en dernier

nécessité de mémoriser l'état de l'union

par exemple, l'union peut être placée dans une structure dont un champ indique le composant actif de l'union :

```
enum Forme { cartesienne, polaire } ;

struct Complex
{
  enum Forme forme ; /* indicateur de la forme courante */

  union
  {
    struct { float re,im ; } car ; /* forme cartésienne */
    struct { float rho,theta ; } pol ; /* forme polaire */
  } valeur ;
} ;
```

l'initialisation peut se faire comme :

```
struct Complex z ;
z.forme = cartesienne ;
z.valeur.car.re = 10.0 ;
z.valeur.car.im =  0.5 ;
```

ou comme :

```
z.forme = polaire ;
z.valeur.pol.rho   = 10.0 ;
z.valeur.pol.theta = 3.14159 ;
```

le module d'un complexe est alors calculé par la fonction :

```
float module(struct Complex z)
{
  float result ;

  switch ( z.forme )
  {
    case cartesienne :
      result = sqrt(z.valeur.car.re * z.valeur.car.re
                    + z.valeur.car.im * z.valeur.car.im) ;
      break ;

    case polaire :
      result = z.valeur.pol.rho ;
      break ;

    default :
      erreur() ;
  }
  return result ;
}
```


16. Les opérateurs de manipulation de bits

`&` `|` `^` `~` `<<` `>>`

exp1`&`*exp2* : chaque bit du résultat vaut 1 si les bits de même rang dans *exp1* et *exp2* valent 1, 0 sinon

exp1`|`*exp2* : chaque bit du résultat vaut 0 si les bits de même rang dans *exp1* et *exp2* valent 0, 1 sinon

exp1`^`*exp2* : chaque bit du résultat vaut 0 si les bits de même rang dans *exp1* et *exp2* ont la même valeur, 1 sinon

`~`*exp* : chaque bit du résultat a une valeur inverse de celle qu'il a dans *exp*

exp1`<<`*exp2* est la valeur de *exp1* décalée de *exp2* positions vers la gauche

exp1`>>`*exp2* est la valeur de *exp1* décalée de *exp2* positions vers la droite

```
1 << 2

7 >> 1
```

17. La compilation conditionnelle

directive adressée au préprocesseur d'insérer ou d'éliminer des lignes de texte source suivant une condition :

```
#if expression_constante
   groupe_de_lignes_1
#endif


#if expression_constante
   groupe_de_lignes_1
#else
   groupe_de_lignes_2
#endif
```

ces groupes de lignes sont du texte quelconque, y compris des directives


```
#define SYSTEME VAX
```

peut précéder :

```
#if SYSTEME == PDP11
  #define BUFSIZE 262144L
  long Size(void) ;
#endif

#if SYSTEME == VAX
  #define BUFSIZE 65535
  unsigned Size(void) ;
#endif

#if SYSTEME == PC_386
  #define BUFSIZE 1024
  int Size(void) ;
#endif
```

ce qui peut aussi s'écrire :

```
#if SYSTEME == PDP11
  #define BUFSIZE 262144L
  long Size(void) ;
#else
  #if SYSTEME == VAX
    #define BUFSIZE 65535
    unsigned Size(void) ;
  #else
    #if SYSTEME == PC_386
      #define BUFSIZE 1024
      int Size(void) ;
    #endif
  #endif
#endif
```


ou, en utilisant la directive `#elif` (équivalente à `#else #if`) :

```
#if SYSTEME == PDP11
  #define BUFSIZE 262144L
  long Size(void) ;
#elif SYSTEME == VAX
  #define BUFSIZE 65535
  unsigned Size(void) ;
#elif SYSTEME == PC_386
  #define BUFSIZE 1024
  int Size(void) ;
#endif
```

construction couramment utilisée :

```
#if 0
  ...
  /* lignes de code à supprimer temporairement */
  ...
#endif
```

L'opérateur defined

defined *symbole*

defined (*symbole*)

vaut 1 si le symbole existe, 0 sinon

exemple :

```
#define VAX
```

peut précéder :

```
#if defined PDP11
  #define BUFSIZE 262144L
  long Size(void) ;
#endif

#if defined VAX
  #define BUFSIZE 65535
  unsigned Size(void) ;
#endif

#if defined PC_386
  #define BUFSIZE 1024
  int Size(void) ;
#endif
```


ce qui peut s'écrire aussi, en utilisant la directive `#ifdef`, équivalente à `#if defined`, comme :

```
#ifdef PDP11
  #define BUFSIZE 262144L
  long Size(void) ;
#endif

#ifdef VAX
  #define BUFSIZE 65535
  unsigned Size(void) ;
#endif

#ifdef PC_386
  #define BUFSIZE 1024
  int Size(void) ;
#endif
```

il existe aussi une directive `#ifndef` équivalente à `#if !defined`

`defined` peut être utilisé avec la directive `#undef` pour redéfinir un symbole :

```
#if defined(PI)
    #undef PI
#endif
#define PI acos(-1)
```

18. Les macros

18.1. Définition et invocation

macro = symbole dont la valeur est paramétrable

```
#define SIGNE(X) X<0 ? -1 : (X>0 ? 1 : 0 )
```

l'invocation de la macro est similaire à l'appel d'une fonction :

```
int s, z=5 ;
s = SIGNE(z) ;
```

expansé en :

```
s = z<0 ? -1 : (z>0 ? 1 : 0 ) ;
```

gain : temps et code qui auraient été nécessaires à l'appel d'une fonction

la macro est développée à chaque invocation

18.2. Macros et fonctions

certaine souplesse par rapport aux fonctions

- la macro `SIGNE` fonctionne pour tout type accepté comme opérande de `<`
- les mots clés peuvent être manipulés :

```
#define MALLOC(NB,TYPE) (TYPE*) malloc(NB*sizeof(TYPE))

ptr = MALLOC(100,int) ;
```

comparaison fonction/macro :

```
#define ALLOUER(PTR,NB,TYPE) \
   if ((PTR=malloc(NB*sizeof(TYPE)))==NULL ) erreur()

void allouer(void **ptr, int nb, int taille)
{
  if ( (*ptr=malloc(nb*taille)) == NULL ) erreur() ;
}
```

utilisation de la macro :

```
int *ptr ;
ALLOUER(ptr,100,int) ;
```

utilisation de la fonction :

```
int *ptr ;
allouer(&ptr,100,sizeof(int)) ;
```

18.3. Les symboles prédéfinis

__LINE__ est substitués par le numéro de ligne courante

__FILE__ est substitués par le nom du fichier source courant

```
#define ERREUR(msg) printf("fichier %s ligne %d : %s",\
                           __FILE__,__LINE__,msg)
```

18.4. Les caractères spéciaux

caractère

le caractère # appliqué à un argument l'entoure de guillemets :

```
#define DUMP(x) printf("%s == %d",#x,x) ;
a = 1 ;
DUMP(a) ;
```

affiche à l'exécution "a == 1"

C dispose d'un moyen de contrôle dynamique d'assurer qu'une condition reste toujours vérifiée à un endroit donné du programme :

```
assert(indice<=limite) ;
```

Si indice est supérieur à limite lors de l'évaluation de l'assert, l'exécution est interrompue et un message est envoyé :

```
Assertion failed:
           indice<=limites, file ESSAI.C, line 122
```

ce contrôle peut être désactivé par la définition du symbole NDEBUG

assert est définie dans le fichier assert.h comme :

```
#ifndef NDEBUG
  static char msg[] = "Assertion non vérifiée : %s, "
                      "fichier %s, ligne %d";
  #define assert(exp)                                    \
  {                                                      \
    if (!(exp))                                          \
    {                                                    \
      fprintf(stderr,msg,#exp,__FILE__,__LINE__);        \
      fflush(stderr); abort();                           \
    }                                                    \
  }
#else
  #define assert(exp)
#endif
```

caractères

permettent de concaténer deux éléments pour n'en former qu'un seul :

```
#define VAR(nom,indice) nom##indice
VAR(A,1) = 1 ;
VAR(A,2) = 2 ;
VAR(A,3) = 3 ;
```

sera expansé en :

```
A1 = 1 ;
A2 = 2 ;
A3 = 3 ;
```


19. Les entrées/sorties

les entrées/sorties ont pour support les flux, dont le type est `FILE`

`FILE` est défini dans le fichier `stdio.h`

`FILE` correspondant à une structure dont les champs sont utilisés en interne par les fonctions d'entrées/sorties de la bibliothèque standard

19.1. Les flux prédéfinis

trois flux sont prédéfinis :

- `stdin` : l'entrée standard
- `stdout` : la sortie standard
- `stderr` : la sortie d'erreur standard

à ces flux sont associés des fichiers ou des périphériques au moment du lancement

les affectations par défaut sont :

- le clavier pour le flux `stdin`
- l'écran pour le flux `stdout`
- l'écran pour le flux `stderr`

19.2. Les fonctions d'entrées/sorties

➢ fonction getc{XE "getc"}

```
FILE *f ;
char c ;
...
c = getc(f) ;                          /* lit un caractère */
```

➢ fonction fgets{XE "fgets"}

```
FILE *f ;
char s[80] ;
...
fgets(s,79,f) ;                         /* lit une chaîne */
```

➢ fonction fscanf{XE "fscanf"}

```
FILE *f ;
char s[80] ;
int i ;
...
fscanf(f,"%s %d",s,&i) ;   /* lit des données formatées */
```

➢ fonction fputc{XE "fputc"}

```
FILE *f ;
char c ;
...
fputc(c,f) ;                        /* écrit un caractère */
```


➢ fonction fputs{XE "fputs"}

```
FILE *f ;
char s[80] ;
...
fputs(s,f) ;                           /* écrit une chaîne */
```

➢ fonction fprintf{XE "fprintf"}

```
FILE *f ;
char s[80] ;
int i ;
...
fprintf(f,"%s %d",s,i) ; /* écrit des données formatées */
```

➢ fonction fread{XE "fread"}

```
FILE *f ;
void *ptr ;
...
fread(ptr,128,1,f) ;          /* lit des données brutes */
```

➢ fonction fwrite{XE "fwrite"}

```
FILE *f ;
void *ptr ;
...
fwrite(ptr,128,1,f) ;       /* écrit des données brutes */
```

➢ fonction fseek{XE "fseek"}

```
FILE *f ;
long pos ;
...
fseek(f,pos,SEEK_SET) ;
fseek(f,pos,SEEK_CUR ;
fseek(f,pos,SEEK_END) ;
```

➢ fonction ftell{XE "ftell"}

```
FILE *f ;
long pos ;
...
pos=ftell(f) ;
```


➢ fonctions fopen et fclose{XE "fopen"}{XE "fclose"}

```
FILE *f ;                                /* variable flux */
int i=0, lu ;

f = fopen("ESSAI.DAT","r") ;       /* ouverture du flux */
if ( f==NULL ) erreur() ;

do {
                                  /* lecture de NB blocs */
     lu = fread(tab[i],TAILLE,NB,f) ;
     i++ ;
                  /* tant que la lecture se passe bien */
   } while ( lu==NB ) ;
fclose(f) ;                          /* fermeture du flux */
```

l'exemple suivant lit un fichier texte et réécrit les données dans un fichier binaire

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

#define N 100
#define L 40

typedef struct { int a ;
                 float x ;
                 char s[L] ;
               } ELEMENT ;

const char SOURCE[] = "TEXTE1" ;
const char CIBLE[] = "BINAIRE" ;

void lire(ELEMENT[], int *) ;
void ecrire(ELEMENT[], int) ;
void traiter(ELEMENT[], int) ;

int main()
{
  ELEMENT tab[N] ;
  int n ;

  lire(tab,&n) ;
  traiter(tab,n) ;
  ecrire(tab,n) ;

  return 0 ;
}
```



```
void lire(ELEMENT tab[], int *nb)
{
  int i ;
  FILE *f ;

  f = fopen(SOURCE,"r") ; assert(f!=NULL) ;

  i = 0 ;
  while ( fscanf(f,"%d %f %s\n",
    &tab[i].a, &tab[i].x, &tab[i].s) == 3 )
  {
    i++ ;
    assert(i<N) ;
  }

  fclose(f) ;
  *nb = i ;
}

void ecrire(ELEMENT tab[], int nb)
{
  FILE *f ;

  f = fopen(CIBLE,"w") ; assert(f!=NULL) ;
  fwrite(tab, sizeof(ELEMENT), nb, f ) ;
  fclose(f) ;
}


void traiter(ELEMENT tab[], int nb)
{
  /* ... */
}
```

20. Les classes d'allocation

la classe d'allocation détermine pour un identificateur :

- sa durée de vie
- sa portée
- son emplacement

quatre classes d'allocation :

- `auto`
- `static`
- `extern`
- `register`

20.1. La classe auto

Une variable automatique :

- est créée lors de l'entrée dans le bloc et est détruite à la sortie
- son éventuelle initialisation est réalisée à l'exécution
- n'est visible que dans son bloc de définition et dans ses sous-blocs
- son emplacement est généralement sur la pile

la classe `auto` est seulement autorisée pour les variables locales

c'est la classe par défaut de toute variable locale

```
void f(void)
{
  auto int i ;
  ...
}
```

20.2. La classe static

Une variable statique :

- est créée et éventuellement initialisée à la compilation
- sa portée est au plus le fichier source comportant sa définition
- son emplacement est généralement dans le segment des données

```
void f(void)
{
  static int initialise=0 ;

  if ( ! initialise )
  {
    /* traitement initial */
    initialise=1 ;
  }
  ...
}
```


pour une fonction, la classe `static` indique que la fonction n'est visible que dans le fichier source comportant sa définition

il est ainsi possible d'avoir dans un même programme plusieurs fonctions de même nom, à partir du moment où :

- elles sont définies dans des fichiers sources distincts
- et si toutes, sauf éventuellement une, sont de classe `static`

```
static void erreur(void)
{
  ...
}
```

20.3. La classe extern

une variable externe :

- est créée et éventuellement initialisée à la compilation
- elle est visible dans son fichier de définition, et dans tout autre fichier s'il comporte une déclaration `extern` d'une variable de même nom
- son emplacement est généralement dans le segment des données

c'est la classe par défaut de toute variable globale :

```
#include <stdio.h>
int x ;
main()
{
  ...
}

#include <stdio.h>
extern int x ;
main()
{
  ...
}
```

une fonction externe est une fonction visible de tout fichier

la classe `extern` est la classe par défaut des fonctions :

```
extern char *gets(char *) ;
```

est une déclaration équivalente à

```
char *gets(char *) ;
```


20.4. La classe register

la classe `register` est utilisée pour définir des variables locales ou des paramètres

elle demande au compilateur de minimiser le temps d'accès

seuls certains types peuvent être de classe `register`

le nombre d'objets de classe `register` est limité à un instant donné

les objets supplémentaires sont considérés alors de classe `auto`

cela ne nuit pas à la portabilité :

```
void f(void)
{
  register int i ;

  for ( i=0 ; i<10000 ; i++ )
  {
    ...
  }
}
```

20.5. Récapitulation

classe	création	durée de vie	portée	emplacement
auto	exécution	bloc	bloc	pile
static	compilation	programme	bloc ou fichier	donnée
extern	compilation	programme	globale	donnée
register	exécution	bloc	bloc	registre

identificateur	classe par défaut	classes autorisées
variable globale	extern	static
variable locale	auto	static register
fonction	extern	static