Cours de Programmation 3

Introduction au langage C

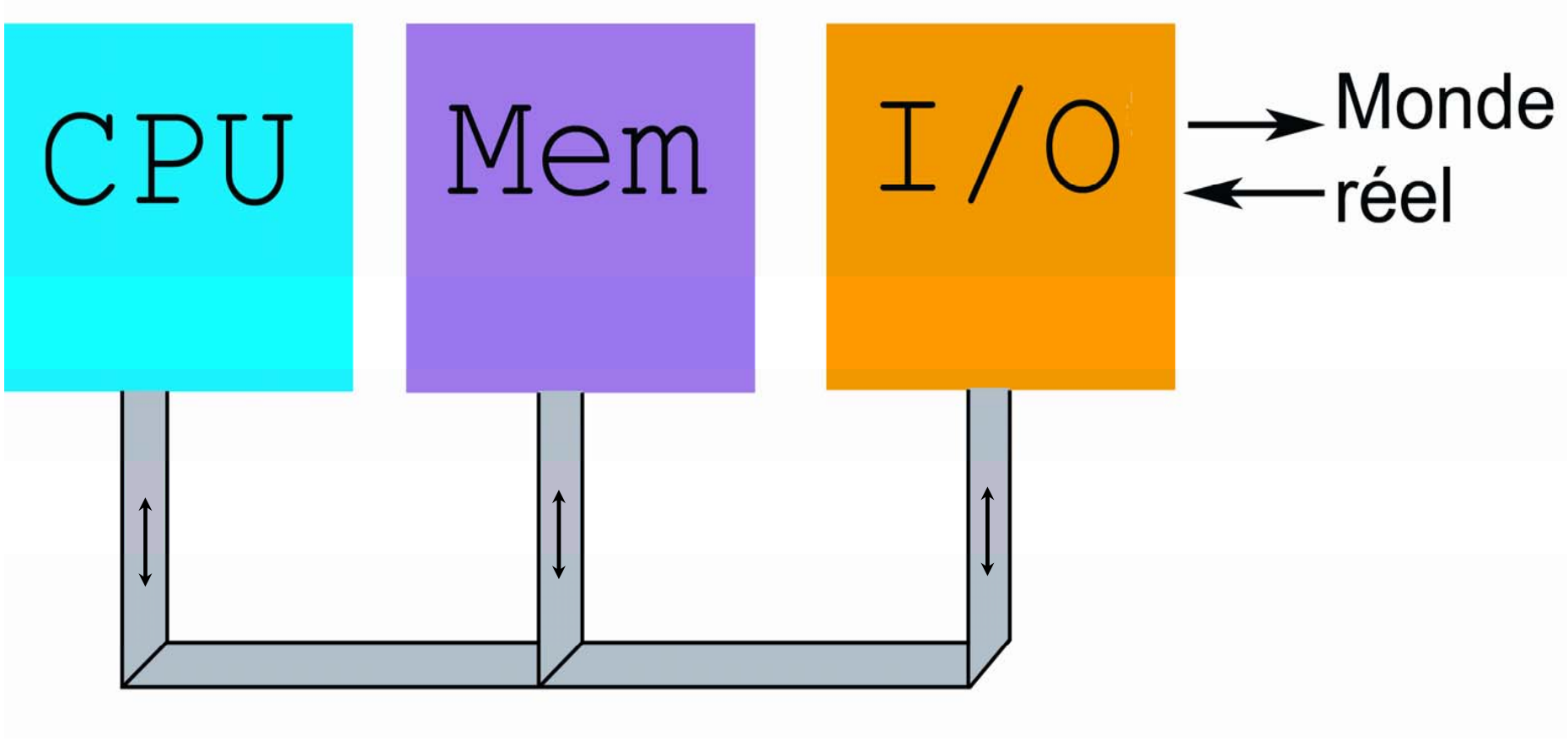
Roger D. Hersch

Laboratoire de Systèmes Périphériques (LSP)

EPFL

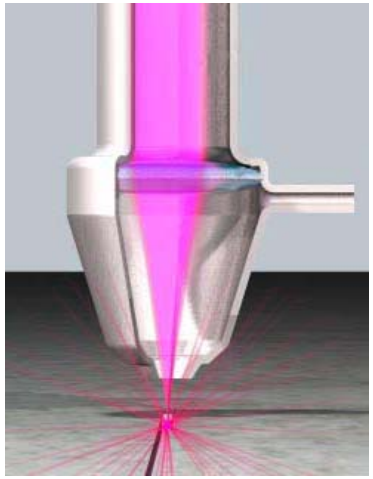
© LSP-EPFL 2005

La programmation, pourquoi?



La programmation, pourquoi?

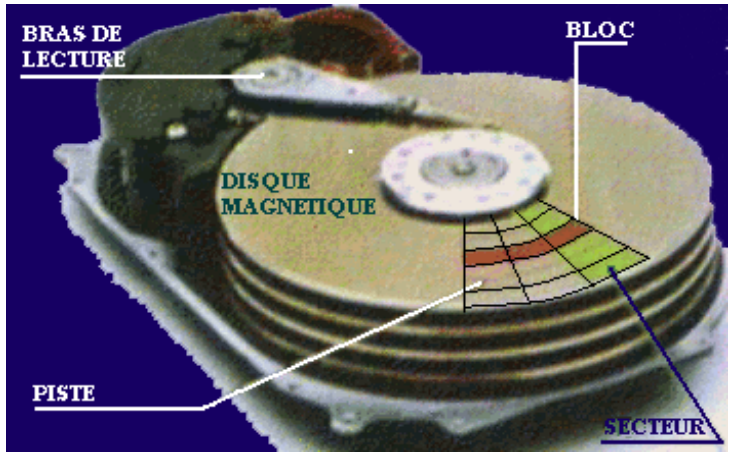
Le programme en exécution est le « cerveau» qui contrôle un dispositif mécanique, électrique, optique, chimique, etc …



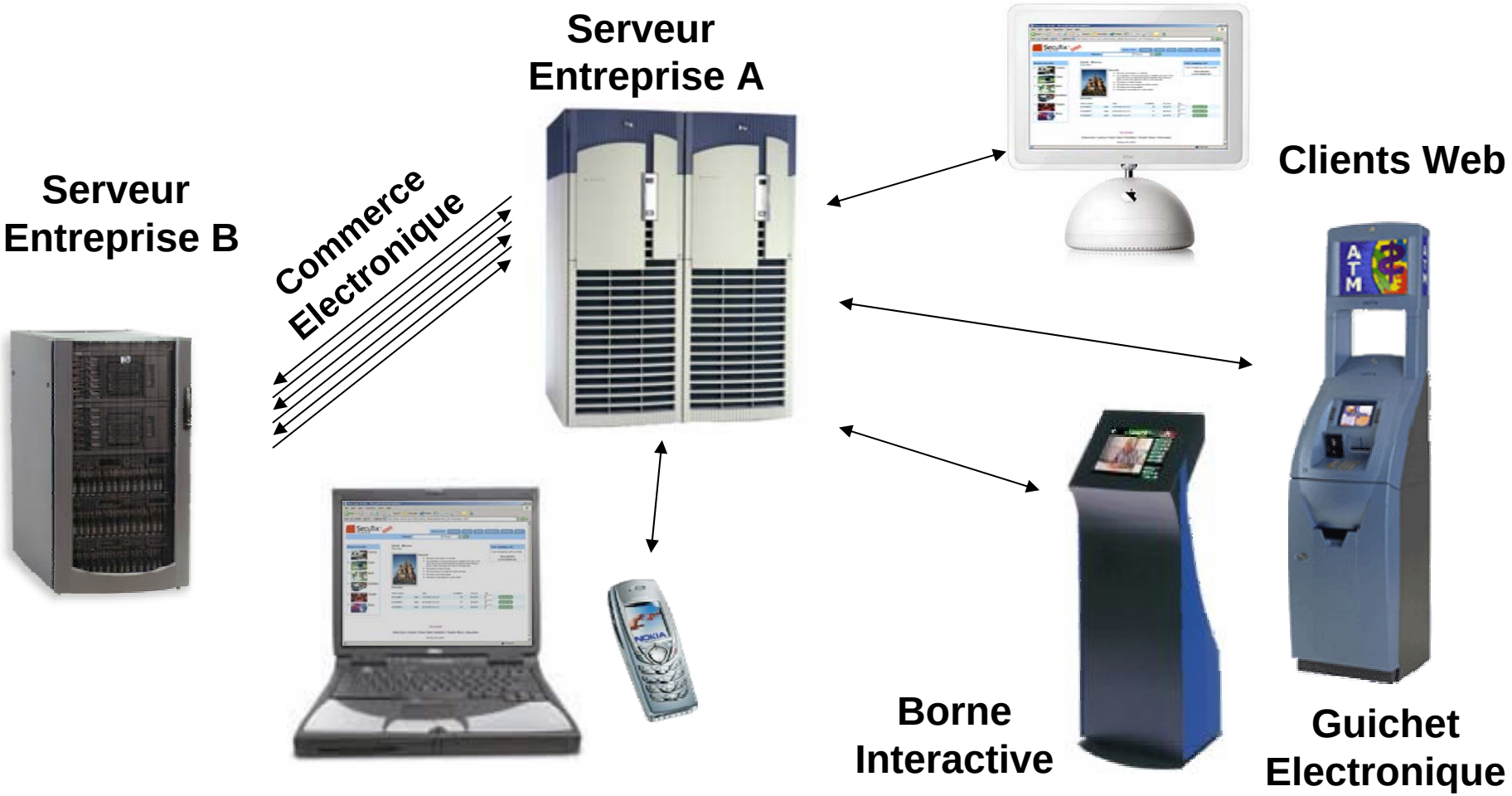
…

Microprocesseurs

On trouve des microprocesseurs programmés aussi dans:



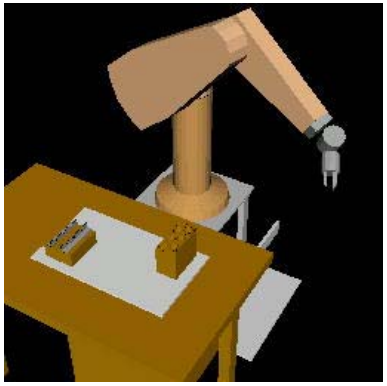
Applications en ligne – e-commerce

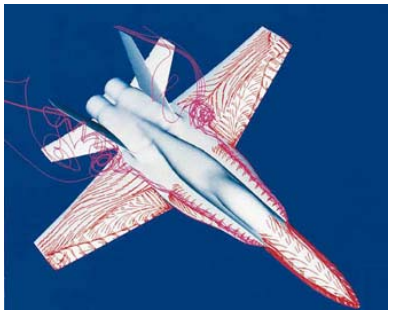


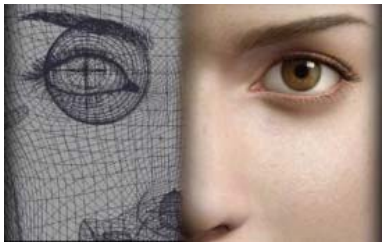
Les ordinateurs sont omniprésents dans les banques, administrations et entreprises.

Autres domaines

- Informatique industrielle
  - Systèmes de contrôle
- Informatique embarquée
  - Voitures, avions, aérospatiale
- Aide à la conception
  - Modélisation
  - Simulation
- Image de synthèse







Domaines avancés

- Ubiquitous computing (processeurs cachés dans les parois, les appareils, etc..) ?
- Informatique intelligente: vision par ordinateur, agents intelligents, dictée automatique, traduction automatique, déduction automatique de savoir à partir d'observations, etc.. ?
- Informatique pour personnes handicapées ?
- Usines automatisées à 100% ?
- L'informatique redondante et auto réparatrice ?

La programmation: les défis

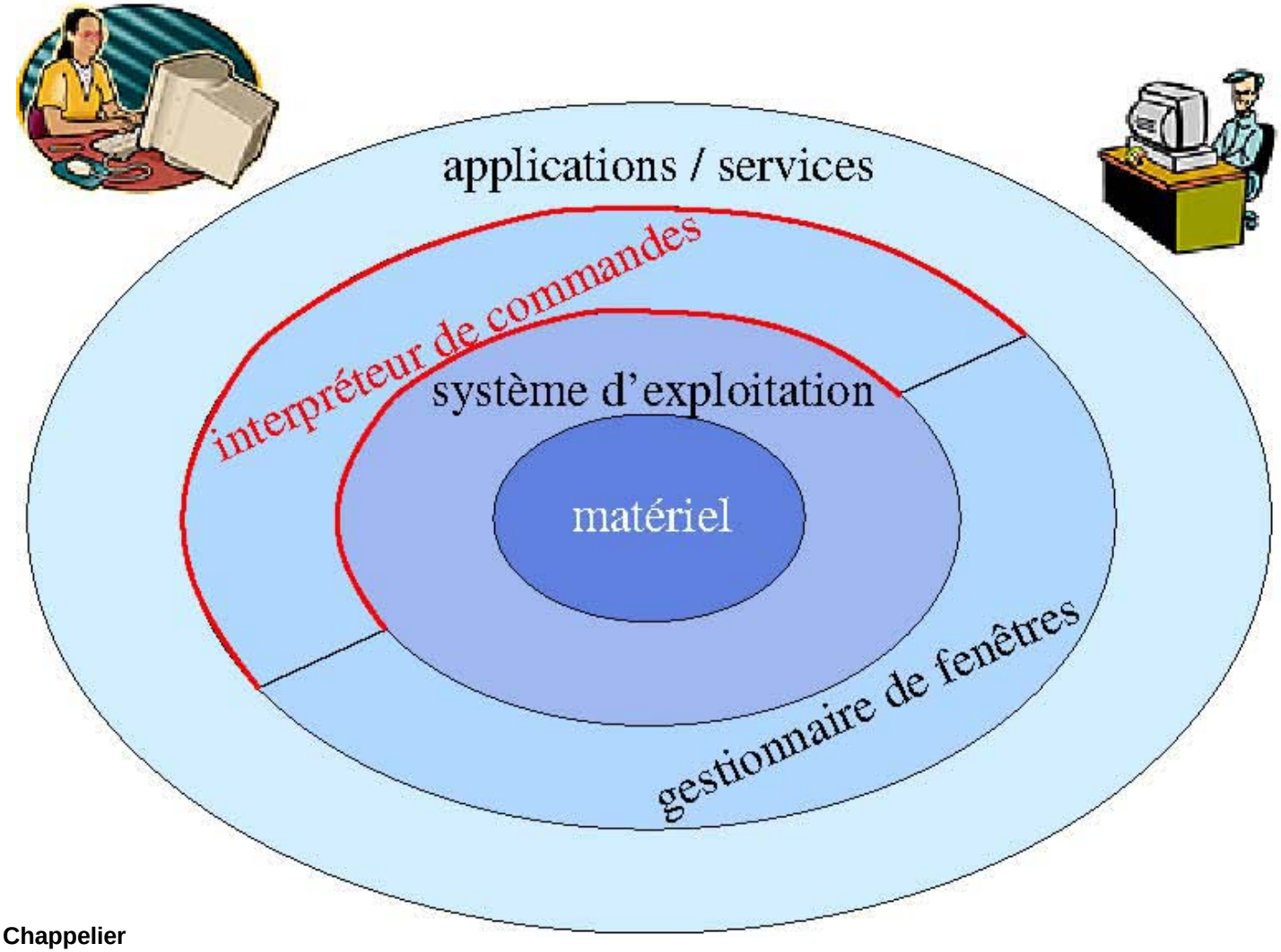
1. Créer des programmes portables et transférables d'un programmeur à un autre:
   - Les fonctions (classes en C++) isolent un élément de fonctionnalité.
   - Découpage en modules et compilation séparée.
   - Documentation de chaque fonction (classe).
   - Documentation globale.
2. Débusquage d'erreurs: simple pour petits programmes, mais croît de manière exponentielle avec la taille.
   - Découper en briques élémentaires.
   - Tester chaque brique séparément.
3. Développement par un groupe de programmeurs:
   - Spécification des interfaces.
   - Programmation C++: interfaces souples

Erreurs dans les programmes ➔ catastrophes

Le programme est votre miroir!

Ai-je l’esprit clair et logique?

Interpréteur de lignes de commandes UNIX



Source: J.C. Chappelier

Shell

Le shell est un programme qui s'exécute et attend des commandes provenant de l'utilisateur.

Commandes fréquentes:

- `ls` — **listing des fichiers & répertoires présents**
- `ls *.txt` — **listing des fichiers se terminant par «.txt»**
- `ls –l` — **listing détaillé des fichiers**

- `gcc hello.c –o hello` — **compilation du programme hello.c en fichier exécutable hello**
- `g++ hello.cpp –o hello` — **Idem, mais pour fichier .cpp**
- `./hello` — **exécution du fichier hello**

- `more fileName.txt` — **affichage du texte se trouvant dans fileName.txt**
- `cat fileName.txt` — **affichage du texte se trouvant dans fileName.txt**

Commandes shell (suite)

- `pwd` — affichage du nom du répertoire courrant
- `cd nomRepertoire` — le répertoire courrant devient nomRepertoire
- `cd ..` — remonter d'un répertoire
- `mkdir directoryName` — créer un nouveau répertoire
- `mv fileName1 fileName2` — modifier le nom du fichier
- `cp fileName3 fileName4` — copier fileName3 dans fileName4
- `find . –name "*prog*"` — trouver dans l'arborescence (.) le nom de fichier qui contient la chaîne de caractères *prog*
- `grep 'monNom' monFichier.txt` — recherche 'monNom' dans monFichier.txt
- `grep 'monNom' *` — recherche 'monNom' dans tous les fichiers

Commandes shell (suite)

- `ls –l > maListeFichiers` **redirection vers un fichier**

- `emacs &` **appel de l'éditeur emacs**
- `emacs` **appel de l'éditeur emacs sans revenir**
- `ctl-z` **suspend emacs et revient au shell**
- `bg` **exécute emacs en arrière-fond**
- `fg` **exécute emacs en foregroung**

Editeur emacs

Commandes fréquentes:

`ctl-x ctl-f`	**ouvrir fichier**
	se déplacer dans le fichier avec curseur
	sauver fichier
	sauver fichier sous le nom
`alt-w`	**copier(chaîne de caractères)**
`ctl-w`	**couper(chaîne de caractères)**
`ctl-y`	**coller(chaîne de caractères)**
`ctl-shift-_`	**undo**

options: afficher syntaxe C, C++

Le langage C

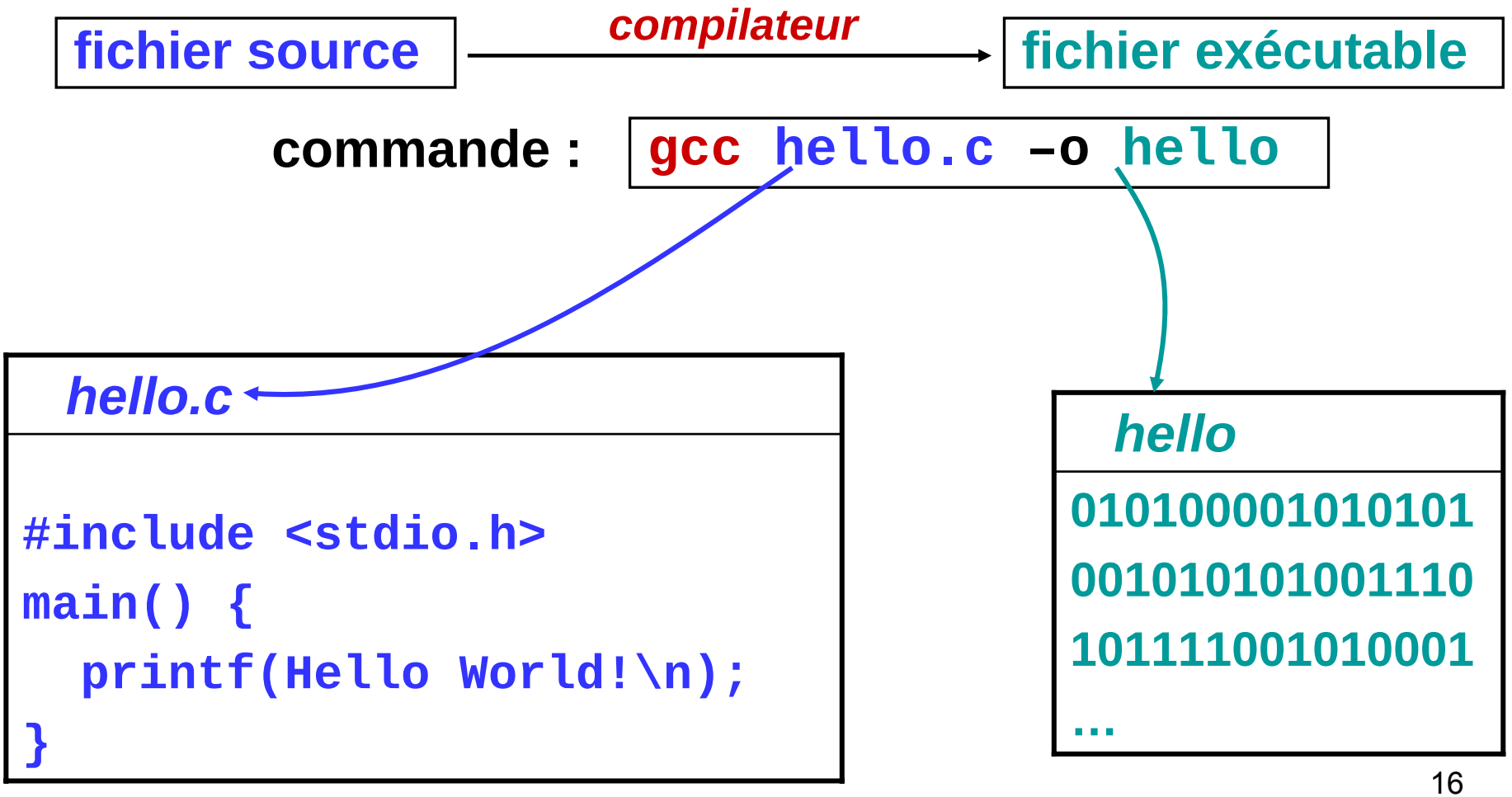
Le langage C est un langage typé impératif compilé.

- **un langage compilé => applications efficaces.**
- **un langage proche du matériel informatique => mémoire accessible par des pointeurs.**
- **un langage rapide.**
- **un langage disponible sur toutes les plateformes avec des librairies standardisées (stdlib.h, stdio.h, string.h, math.h, time.h, …)**

Question:

Quels sont les avantages & inconvénients d'un langage compilé par rapport à un langage interprété?

Compilation d'un programme C



Source: J.C. Chappelier

C, C++ & Java

	C	C++	Java
Langage	impératif	objet	orienté objet
Code objet	code machine		byte code
Vérification à la compilation	faible	forte	forte
Allocation mémoire	statique et dynamique (par programmeur)		dynamique (mais non explicite)
Accès direct de la mémoire	oui (pointeurs)		non
Vérification des accès mémoire	non	possible	oui
Désallocation	à la main	partiellement automatique (par librairies)	automatique (garbage collector)
Rapidité d'exécution	élevée		moyenne
Malléabilité des interfaces entre modules	faible	forte	moyenne

Source: J.C. Chappelier

Structure générale d'un programme C

La structure très générale d'un programme C est la suivante:

```
#include <librairies.h>
...

déclaration d'objets globaux)                    [ à éviter ]

Déclarations de fonctions utiles                 [ recommandé ]

int main()
{
corps du
programme principal                    [ si possible assez court ]
return 0;
}
```


Source: J.C. Chappelier

Algorithme à programmer

Résoudre (dans $\mathbb{R}$) une équation du second degré de type:

$$ax^2 + bx + c = 0$$

saisir les données a, b et c

$$\Delta \leftarrow b^2 - 4ac$$

si $(\Delta < 0)$ **afficher « pas de solution »;**

sinon si $(\Delta = 0)$ $x \leftarrow -\frac{b}{2a}$

afficher x;

sinon $x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

afficher x_1, x_2

Source: J.C. Chappelier

Programme C correspondant

```
#include <stdio.h> //quadraticEq.c
#include <math.h>
int main() {
   double a     = 1.0;
   double b     = 0.0;
   double c     = 0.0;
   double delta = 0.0;

   printf("Entrez b : "); scanf("%lf", &b);
   printf("Entrez c : "); scanf("%lf", &c);
   delta = b*b - 4*a*c;
   if (delta < 0.0) {
     printf("pas de solutions réelles\n");
   } else if (delta == 0.0) {
     printf("une solution unique : %f\n", -b/(2.0)*a);
   } else {
     printf("deux solutions: %f et %f\n", (-b-sqrt(delta))/(2.0*a),
                    (-b+sqrt(delta))/(2.0*a));
   }
}
```

données

traitements

structures de contrôle

Source: J.C. Chappelier

Exemple d'exécution

Console:

```
X:\prog3\Debug>eq_second_degre
Entrez b : 4
Entrez c : 4
une solution unique : -2.000000

X:\prog3\Debug>eq_second_degre
Entrez b : -12
Entrez c : 6
deux solutions: 0.522774 et 11.477226
```

Types et variables: Types de base

En langage C, les variables sont déclarées selon la séquence `<type variable>` ou `<qualificateurDeType type variable>`. Il est possible d'associer une valeur initiale à une variable lors de sa déclaration.

Exemples:

```
                        /* typesBase.c */
char   monCaractere;    /* type caractère */
int    maValeur = 8;    /* type entier 16 ou 32 bits */
unsigned char entier8bits;    /* type entier 8 bits */
short int     entier16bits;   /* type entier 16 bits */
float  monReel32;       /* type virgule flottante 32 bits*/
double monReel64;       /* type virgule flottante 64 bits*/
```

Exemples

Spécificateur de Type	taille (bits)	Signe
unsigned char monOctet = 0x7F ;	8	%u ou %X
int monEntier = 1525 ;	32 (ou 16)	%d ou %6d
short int	16	%d
long int	32	%l
float monPoids = 75.54	32	%f ou %6.2f
double	64	%lf
char car[80] = "maChaine" ;	8	%c

Une chaîne de caractères est toujours terminée par l'élément \0.
La chaîne ci-dessus est donc "maChaine\0".

La taille des types dépend de la machine.
Pour savoir: `sizeof(type);`

Définition d'un nouveau type

Exemple:

```
typedef identificateurType nouvelIdentificateur;
```

```
typedef int entier;
entier distance;
          /* variable distance de type int */
```

Changement de type

Le changement de type s'effectue en plaçant le nouveau type devant la variable:

```
maVariable = (nouveauType) variable;
```

Exemple:

```
float x;
int a = 9, b = 2;
x = (float) a/b;      /* x vaut 4.5 */
```

Affichage de chaînes de caractères

```
#include<stdio.h>  // printf.c
int main( )
{
  int entier; float valeur; char lettre;
  int* pointeur;
  entier = 435; valeur = 136.25; lettre = 'A';
   pointeur = &entier;
  printf("%d\n",entier);       //    435
  printf("%5d\n",entier);      //    □□435
  printf("%-5d\n",entier);     //    435□□
  printf("%5d\n",-entier);     //    □ -435
  printf("%2d\n",entier);      //    435
  printf("%f\n",valeur);       //    136.250000
  printf("%10.2f\n",valeur);   //    □□□□136.25
  printf("%2.6f\n",valeur);    //    136.250000
  printf("%6c\n",lettre);      //    □□□□□A
  printf("%p\n", pointeur);    //    0012FF7C
  printf("%d\n", *pointeur);   //    435
}
```

Codes ASCII

Codes ASCII								
H'	0	10	20	30	40	50	60	70
0	NUL	DLE		0	@	P	‘	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	¨	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BELL	ETB	’	7	G	W	g	w

ISOLatin1	
E0	F0
à	>
á	ñ
â	ò
ã	ó
ä	ô
å	õ
æ	ö
ç	÷

Codes ASCII (2)

Codes ASCII								
H'	0	10	20	30	40	50	60	70
8	BS	CAN	(	8	H	X	h	x
9	TAB	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

ISOLatin1	
E0	F0
è	ø
é	ù
ê	ú
ë	û
ì	ü
í	‡
î	
ï	ÿ

Lecture du clavier

La routine de lecture du clavier permet de lire plusieurs variables tapées au clavier les unes après les autres. De manière similaire à la routine d'affichage `printf`, la routine `scanf` nécessite un spécificateur de type placé dans la chaîne. Les paramètres à écrire par les valeurs tapées au clavier sont transmis par référence, c'est-à-dire par leur adresse.

```
int scanf(chaineDeFormat, &arg1, &arg2,..,&argn);
```

La fonction `scanf` retourne le nombre d'éléments lus.

Par exemple, pour lire une succession de trois caractères telle que "x y" (x, espace, y), on peut lire les 3 caractères par l'appel de routine :

```
char a, b, c;  /* trois caractères */
scanf ("%c%c%c", &a, &b, &c); // &: adresse
                              // de la variable
```

Lecture du clavier (2)

Pour lire une chaîne de caractères, définie en C comme un tableau de caractères, on déclare d'abord le tableau, puis on passe le nom du tableau, c'est-à-dire son adresse comme paramètre à la procédure scanf:

```
/* lectString.c */
#include <stdio.h>
int main ()
{
  char str[80];  // chaîne: tableau de 80 caractères
  printf("Entrez une chaine de caractères \
                          terminée par un espace: \n");
  scanf ("%80s", str); // lit au maximum 80 caractères
            /* str représente le pointeur au tableau*/
  printf("Voici la chaine de caractères: %s\n", str);
}
```

Lecture du clavier (3)

Lecture de caractères par la fonction `getchar()`:

```
#include <stdio.h>              /* lectgetchar.c */
int main () {
      int c = getchar();
      while (c != '\n') {
            putchar(c);
            c = getchar();
      }
}
```

Console:

```
X:\prog3\Debug>lecture_getchar
Bienvenue au cours de programmation 3!
Bienvenue au cours de programmation 3!
```

Question:

Expliquez comment et selon quelle séquence les caractères sont-ils affichés?

Exemples

Lecture d'une ligne de caractères par la fonction `fgets()`:

```
#include <stdio.h>                /* lectfgets.c */
int main () {
  char line[81];
  printf("Input a string: ");
  fgets( line, 80, stdin );
            // get string, max size 80 characters
  printf("The line entered was: %80s\n",line);
}
```

Console:

```
X:\prog3\Debug>lecture_fgets
Input a string: Bonjour, comment allez-vous?
The line entered was: Bonjour, comment allez-vous?
```

Question: comment obtenir chaque caractère de la chaine de caractères ?

Comparaison de chaînes de caractères: **strcmp**

```
#include <stdio.h>                    /* strcmp.c */
#include <stdlib.h>
#include <string.h>
int main () {
  char  line[81];
  int   result;
  printf("To exit properly, type 'quit' \n");
  gets(line);
  result = strcmp (line, "quit");
  if (result == 0) {
     printf ("You exit the program!\n");
     exit(1);          // exits with value 1
  } else {
     printf ("You have typed: %s\n", line);
     exit(0);          // exits with value 0
  }
}
```

Exemples d’exécutions

Console:

```
X:\prog3\Debug>type_quit
To exit, type 'quit'
bonjour
You have typed: bonjour

X:\prog3\Debug>type_quit
To exit, type 'quit'
quit
You exit the program!
```

Recherche d'un jeton connu dans une chaîne de caractères: **strstr**

```
#include <string.h>           // strstr.c,  VC++, RDH
#include <stdio.h>

char mstr[] =   "brown";      // jeton à rechercher
char mstring[]= "The quick brown dog jumps ";       // chaine caractères
char fmt1[] =   "          1         2         3";  // dizaines
char fmt2[] =   "01234567890123456789012345678890";  // pos unites

int main()
{
  char *pdest;
  int  result;
  printf( "String to be searched:\n\n\t%s\n", mstring );
  printf( "\t%s\n\t%s\n\n", fmt1, fmt2 );
  pdest = strstr( mstring, mstr );
                              // recherche chaîne  mstr[] dans mstring[]
  result = pdest - mstring;
  if( pdest != NULL )
    printf( "%s found at position %d starting from 0\n\n", mstr, result);
  else
    printf( "%s not found\n", mstr );
}
```

Exemple d'exécution

Console:

```
X:\Prog3\Debug> strstr
String to be searched:

          The quick brown dog jumps
                    1         2          3
          012345678901234567890123456789

brown found at position 10 starting from 0
```

Segmentation d'une chaîne en jetons: **strtok**

```
#include <string.h> // strtok.c, permet d'obtenir les jetons se
#include <stdio.h>  // trouvant dans une chaîne de caractères, VC++, RDH

char string[] = "A string\tof ,,tokens\nand some  more tokens";
char seps[]   = " ,\t\n";     // separateurs: espace, tab, retour chariot
char * ptrToken;

int main( void )
{
   printf( "%s\n\nTokens:\n", string );
   ptrToken = strtok( string, seps );
                              // sort le pointeur au premier jeton
   while( ptrToken != NULL )
   {
    /* While there are tokens in "string" */
    printf( " %s\n", ptrToken );//Condition pour que fonctionne ?
    /* Get next token: */
    ptrToken = strtok( NULL, seps ); // pointeur au jeton suivant
   }
}

//Comment verifier que strtok modifie string ?
```

Exemple d'exécution

Console:

```
D:\Mathieu\Prog3 - cours C\prog\Debug>strtok
A string        of ,,tokens
and some  more tokens

Tokens:
 A
 string
 of
 tokens
 and
 some
 more
 tokens
```

Verifier si chaîne originale modifiée

Insérer ces 3 lignes à la fin du programme précédent:

```
printf( "\n\nString after: \t%s\n",string );
for (i=0;i<length;i++) printf("%c %2x ",
      string[i], (unsigned char)string[i]);
```

Console:

```
String after:   A
A 41    0 s 73 t 74 r 72 i 69 n 6e g 67
  0 o 6f f 66    0 , 2c , 2c t 74 o 6f
k 6b e 65 n 6e s 73    0 a 61 n 6e d 64
  0 s 73 o 6f m 6d e 65    0   20 m 6d
o 6f r 72 e 65    0 t 74 o 6f k 6b e 65 n
  6e s 73 Press any key to continue
```

Conversion chaîne de caractères vers entier ou flottant: **atoi**, **atof**

```
#include <stdio.h>                    /* conversion_caractere.c */
#include <stdlib.h>
int main() {
   char       line[81];
   double     opFloatingPoint;
   int        opInteger;
   do{
       printf("Input a number : ");
       gets(line);
       opFloatingPoint = atof (line);
                                /* convert to floating-point */
       printf("%s \t float: \t%f\n", line, opFloatingPoint);

       opInteger = atoi (line);     /* convert to integer */
       printf("%s \t integer: \t%d\n", line, opInteger);
       printf("type 'q' to quit or carriage return\n");
   } while (getchar()!='q');         /* type ‘q’ to quit */
}
```

Exemples d'exécutions

Console:

```
X:\prog3\Debug>conversion_caractere
Input a number : 25.60134826
25.60134826      float:          25.601348
25.60134826      integer:        25
type 'q' to quit or carriage return

Input a number : 15
15       float:          15.000000
15       integer:        15
type 'q' to quit or carriage return

Input a number : 128.3
128.3    float:          128.300000
128.3    integer:        128
type 'q' to quit or carriage return
q
```

Vérifier si une chaine de caractères est un nombre

```
int isNumber(char * string)
{
 // Returns true if the given string starts with a number
   // If atof does not return 0,
   //then string begins with a number

 if (0 != atof(string)) // returns floating point number
    return 1;

   // verify if string starts with the digit '0'
 else if (string[0] == '0') // ‘0’ is legal number
        return 1;
      else return 0; // not a number !
}
```

Opérateurs arithmétiques

Affectation :

A une variable peut être affectée une autre variable ou une constante :

```
const int b = 5;
int a;
a = b ;    /* affectation */
```

La valeur qui se trouve à la position mémoire de "b" est lue par le processeur et est transférée à la position mémoire de "a".

Opérateurs arithmétiques :

Opérateur	Opération	Remarques
+	addition	
-	soustraction	
*	multiplication	
/	division	division entière si les 2 opérandes sont de type entier.
%	modulo	seulement pour opérandes de type entier.

Variables booléennes

En C, il n'y a pas de type booléen prédéfini. Une variable entière à 0 correspond à la valeur "faux", toute autre valeur est "vrai". On fait donc toujours les tests sur le cas "faux".
Vous pouvez définir votre propre type booléen :

```
#include <stdio.h>          // bool.c
#define myfalse 0           // 0 toujours "false"
typedef unsigned char mybool;
                            // autres nombres "true"
int main () {
  // assigne maVariable à vrai
  mybool maVariable = !myfalse;

  printf("maVariable = %0x \n", maVariable);
}
```

Console: **maVariable = 1**

Pointeur à une variable et opérateur &

Il est souvent utile de ne pas travailler directement avec une variable, mais avec un pointeur à une variable (l'adresse mémoire de la variable). L'opérateur & permet d'obtenir le pointeur à une variable. Pour déclarer un pointeur à une variable, on fait précéder le nom de la variable par le signe *.

```
int *ptrEntier1, *ptrEntier2;
```

Lors de l'affectation d'une valeur à la variable pointée, on utilise l'opérateur * pour spécifier la variable, c'est-à-dire pour déréférencer le pointeur.

```
#include <stdio.h>
int main() {
  int maVariable;
  int *monPointeur;
  maVariable = 232;
  monPointeur = &maVariable;
                /* monPointeur pointe sur maVariable*/
  printf(" monPointeur = %p \n", monPointeur);
  printf(" *monPointeur = %d \n" , *monPointeur);
}
```

Représentation hexadécimale d'un pointeur mémoire

entier

Exemple d’exécution

Console:

```
X:\prog3\Debug>pointeurs
 monPointeur = 0012FF7C
 *monPointeur = 232
```

Opérateurs relationnels

Opérateur	**Opération**
==	égalité
!=	inégalité
<	plus petit
>	plus grand
<=	plus petit ou égal
>=	plus grand ou égal

Opérateurs logiques

Opérateur	Opération
!	négation logique
&&	ET logique
||	OU logique

Exemples:

```
if ((z != 0.0) && (2*(x-y)/z < 3.0))
        doThis;

if ((i >= 0) || ((x*y > 0.0) && !(j == 2)))
        doThat;
```

Structures de contrôle

En langage C, une instruction terminée par un point virgule est un énoncé. Plusieurs énoncés groupés entre parenthèses { } forment un énoncé composé, équivalent à un seul énoncé.

Le débranchement *if else*

```
/* deviner un nombre */
#include <stdio.h>          // ifelse.c
#include <stdlib.h>
int main () {
  int magic;                /* nombre magique */
  int guess;                /* nombre supposé */
  magic = rand();           /* obtenir un nombre aléatoire */
  printf (" Suggérez un nombre: ");
  scanf ("%d", &guess);      /*lecture d'un nombre entier*/
  if (guess == magic) {
    printf("*** BRAVO ***");
    printf(" %d est le nombre cherché\n", magic);
  } else {
    printf("Faux ");
    if(guess > magic) printf("trop élevé\n");
    else printf("trop bas\n");
  }
}
```

L'alternative (*?* suivi de :)

```
#include <stdio.h>    // plusgrand.c

int main()
{
  int largest, a, b;
  printf("Entrez deux nombres: ");
  scanf("%d %d", &a, &b);
  largest = ( a > b ? a : b );
  printf("Le plus grand des deux est : %d\n",
                                       largest);

}
```

Débranchements multiples: *switch*

Après chaque test (case), il est nécessaire d'insérer l'instruction *break* pour ne pas exécuter les cas suivants.

Exemple :

```
int ch;
ch = getchar();/*lecture d'un nombre du clavier*/
switch(ch)  {
  case ' 1 ' : check_spelling () ; break ;
  case ' 2 ' : correct_errors () ; break ;
  case ' 3 ' : display_errors () ; break ;
  default :
    printf ("No option selected");
                /* si aucun des cas précédents*/
}
```

La boucle *for*

La boucle *for* définit une instruction d'initialisation, une condition de terminaison et une instruction exécutée à chaque fin de boucle.
La condition de terminaison est testée à chaque nouvelle boucle.

Exemples :

(a)

```
#include <stdio.h>
int main () {
  int x ;
  for (x = 1; x <= 100; x++) printf ("%d ", x) ;
}
```

(b)

```
int main () {
  int x,z;
  for (x = 100; x != 65; x = x-5) {
    z = x * x;
    printf ("The square of %d is %d \n", x, z) ;
  }
}
```

La boucle *while*

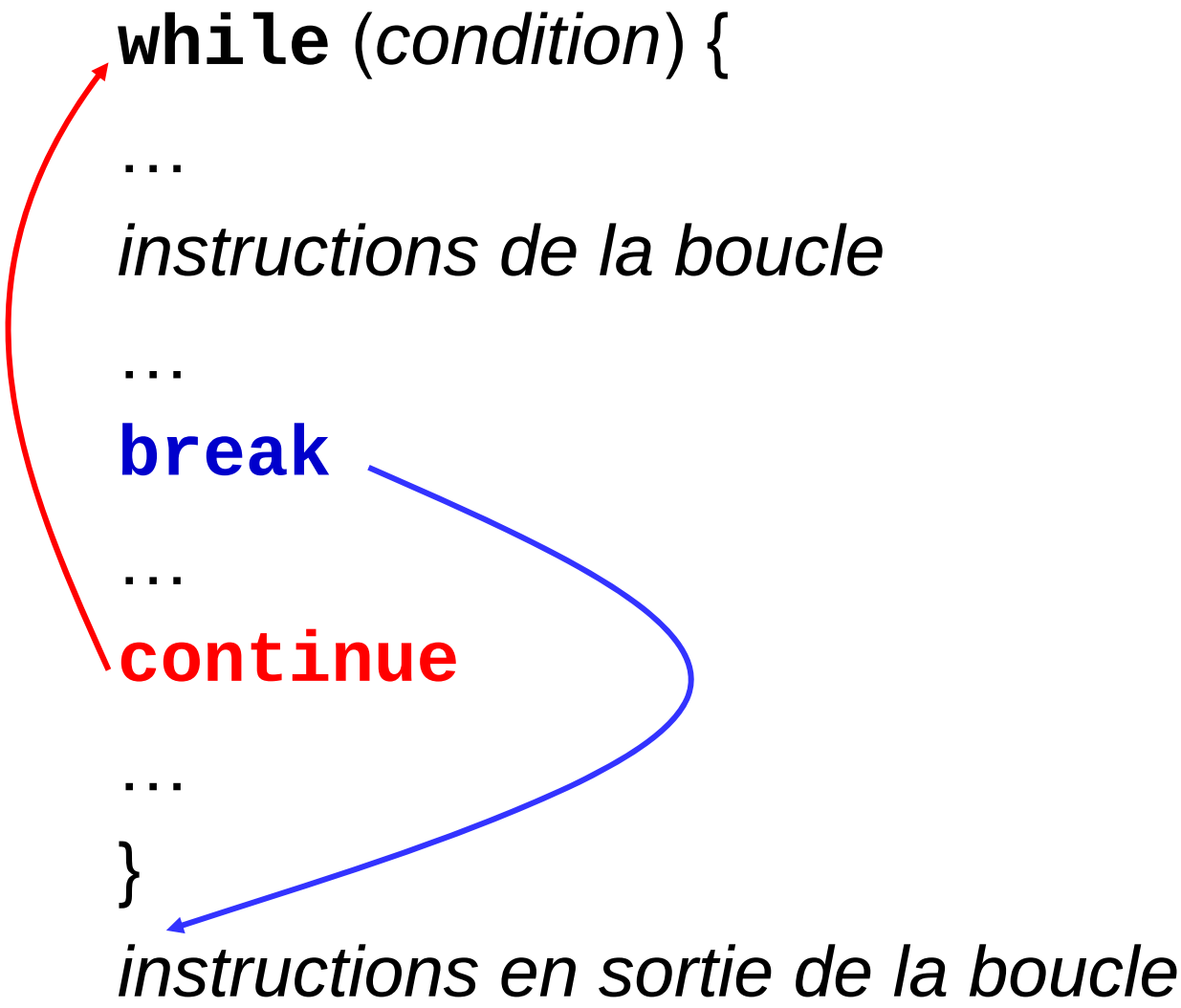
```
/* Affiche une table de conversion Fahrenheit - Celsius */

#include <stdio.h> /* fahrcelsius.c */
#define LOWER 0   /*borne inf de la table des températures*/
#define UPPER 300 /*borne sup de la table des températures*/
#define STEP  20  /* incrément */

int main () {
   int fahr, celsius;    /* déclarations */
   fahr = LOWER;
   printf("Tableau de conversion Fahrenheit vers Celsius\n")

   while (fahr <= UPPER) {
       celsius = 5 * (fahr - 32) / 9;
       printf("%3d %6d\n" , fahr, celsius);
       fahr = fahr + STEP;
   }
}
```

Exemple d’exécution

Console:

```
X:\prog3\Debug> conv_fahr_celsuis
Tableau de conversion Fahrenheit vers Celsius
  0    -17
 20     -6
 40      4
 60     15
 80     26
100     37
120     48
140     60
160     71
180     82
200     93
220    104
240    115
260    126
280    137
300    148
```

Instructions break et continue



Source: J.C. Chappelier

L'expression *do while* et la fonction *exit (0)*

La fonction *exit (0)* permet de quitter un programme.

```
menu () {
  char ch ;
  printf ("1. Vérifier l'orthographe\n") ;
  printf ("2. Corriger les erreurs\n") ;
  printf ("3. Afficher les erreurs\n") ;
  printf ("4. Quitter\n") ;
  printf ("   Entrer votre choix\n") ;
  do {
    ch = getchar () ;    /* lecture d'une touche clavier*/
    switch(ch) {
      case '1' : verifierOrthographe(); break;
      case '2' : corrigerErreurs(); break;
      case '3' : afficherErreurs(); break;
      case '4' : exit (0) ;   /* quitte le programme */
    }                         /* 0: succès, 1:échec*/
  } while (ch!='1' && ch!='2' && ch!='3') ;
}
```

Les pointeurs

La déclaration d'un pointeur se fait selon la syntaxe suivante:

```
type *identificateur
```

Cette instruction déclare un variable de nom `identificateur` et de type pointeur sur une variable de type `type`.

Il existe deux *opérateurs* particuliers en relation avec les pointeurs & et *.

& est l'opérateur qui

retourne l'adresse mémoire de la valeur d'une variable.

*** est l'opérateur qui**

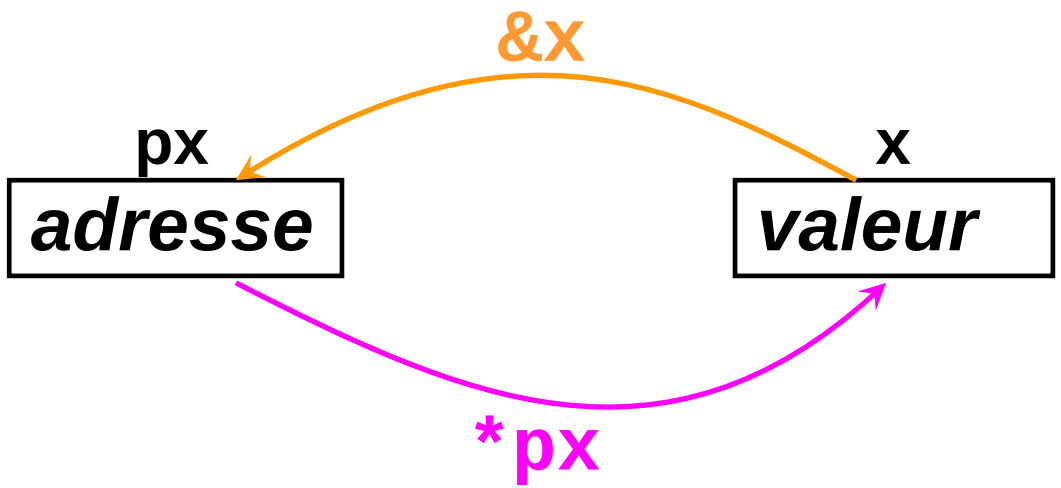
retourne la valeur pointée par une variable pointeur.

Source: J.C. Chappelier

Les pointeurs

Exemple:

```
int x = 3;   /* x est de type entier */
int* px;     /*px est un pointeur sur entier*/

px = &x;     /* px pointe sur la variable x */
printf("%d \n", *px);  /* affiche 3 */
             /*affiche la valeur pointée par px*/
```



Source: J.C. Chappelier

Fonctions et passages de paramètres

```
#include <stdio.h>
void miseAuCarre(int x) //passage de x par copie
{
   x = x * x;
}

int main() {
   int nombre;
   printf("Entrez un nombre: ");
   scanf("%d", &nombre);
                        //reçoit nombre en entrée au clavier
   miseAuCarre(nombre);//appelle miseAuCarre
   printf("affichage nombre: %d \n", nombre);
}
```

Console:

```
Entrez un nombre: 65
affichage nombre: 65
```


Pourquoi n'obtient-on pas le nombre au carré?

Fonctions et passages de paramètres(2)

Les fonctions en C ne permettent de passer des paramètres que par valeur; c'est-à-dire que la valeur passée comme paramètre est copiée à l'intérieur de la fonction puis est utilisée comme variable locale. Le résultat peut être transmis explicitement comme paramètre de retour.

Exemple:

```
#include <stdio.h>
int miseAuCarre(int x)/*paramètre de retour de type entier*/
{
  int xSquare;
  xSquare = x*x;
  return(xSquare);    /*passage du paramètre en retour*/
}
int main() {
  int nbEntree, nbCarre;
  printf("Entrez un nombre: ");
  scanf("%d",&nbEntree);
                      /*reçoit nombre en entrée du clavier*/
  nbCarre = miseAuCarre(nbEntree);
  printf("carré du nombre en entrée: %d \n", nbCarre);
}
```

Exemple d'exécution

Console:

```
X:\prog3\Debug>miseaucarre
Entrez un nombre: 12
carré du nombre en entrée: 144
```

Passages de paramètres par référence

Souvent, il est nécessaire de passer un paramètre par référence, c'est-à-dire que le pointeur à la variable doit être passé et non la variable elle-même. Ceci permet, depuis l'intérieur de la procédure appelée, de modifier une variable se trouvant à l'extérieur de la procédure appelée. Comme le langage C ne permet que le passage de paramètres par copie, le pointeur à la variable doit être passé, puis à l'intérieur de la procédure être utilisé comme un pointeur pour accéder à la variable. L'opérateur & permet d'obtenir le pointeur à une variable.

Exemple:

```
#include <stdio.h>
void miseAuCarre(int x, int * xCarrePtr)
{
  *xCarrePtr = x*x;           /* variable pointée par xCarrePtr */
}
int main() {
  int nbEntree, nbCarre;
  printf("Entrez un nombre: ");
  scanf("%d", &nbEntree);   /*reçoit nombre en entrée du clavier*/
  miseAuCarre(nbEntree, &nbCarre);  /* produit le nombre nbCarre*/
  printf("carré du nombre entré: %d \n", nbCarre);
}
```

Passage de paramètres par référence en C++

En C++, il est possible de passer une variable par référence en utilisant le signe & dans la déclaration des paramètres de la procédure. Ensuite, la variable passée ainsi par référence peut être directement modifiée dans la procédure appelée.

Exemple C++:

```
#include <stdio.h>          // miseAuCarre.cpp
void miseAuCarre(int x, int & xCarre)
                            /* passage de xCarre par référence */
{
  xCarre = x*x;             /* variable xCarre */
}
int main() {
  int nbEntree, nbCarre;
  printf("Entrez un nombre: ");
  scanf("%d", &nbEntree);  /*reçoit nombre en entrée du clavier*/
  miseAuCarre(nbEntree, nbCarre);
  printf("carré du nombre entré: %d \n", nbCarre);
}
```

Passage de paramètres par lignes de commandes

Lors de l'exécution d'un programme C, il est possible de passer des paramètres par la ligne de commande (exécution depuis une console). Ces paramètres sont récupérés dans le programme principal par les deux variables argc et *argv[ ]. La variable argc donne le nombre de paramètres passés. Le tableau de chaînes de caractères *argv[ ] donne l'ensembles des chaînes de caractères passées. L'élément argv[0] donne la première chaîne de caractères, c'est-à-dire le nom du programme exécutable, puis argv[1], argv[2], etc... donne les autres chaînes de caractères.

```
#include <stdio.h>                    //hello.c
#include <stdlib.h>
int main(int argc, char *argv[]) {
  if (argc !=2) { /*si nombre d'arguments pas égal 2*/
    printf("Vous avez oublié de taper votre nom\n");
    exit(0);       /* quitter le programme */
  }
  printf("Hello %s!\n", argv[1]);
        /* affiche la chaîne passée comme paramètre */
}
```

Exemples d'exécutions

Console:

```
X:\prog3\Debug>hello
Vous avez oublie de taper votre nom

X:\prog3\Debug>hello Charles
Hello Charles!
```

Tableaux et pointeurs

En C, les noms de tableaux correspondent à des pointeurs:

- **Ils pointent au premier élément du tableau.**
- **L'espace mémoire requis pour un tableau peut être réservé de manière statique ou dynamique.**
- **Il y a réservation statique, lorsque le nombre d'éléments réservés est connu à la compilation, c'est-à-dire lors de l'écriture du programme.**
- **Il y a réservation dynamique, lorsque le nombre d'éléments du tableau est alloué dynamiquement à l'exécution du programme.**

Tableaux 1D alloué statiquement

L'accès à un élément d'un tableau, par exemple x[3] dans le programme qui suit, est traduit par le compilateur à un accès par pointeur et déréférencement à la position 3 du tableau.

```
#include <stdio.h>  //tableau1Dstat.c
#define NB_ELEM 100
int main() {
  int element3t, element3p;
  int x[NB_ELEM];
        /*réservation d'un tableau de 100 entiers*/
  int t ;
  for(t=0; t < NB_ELEM; t++) x[t] = t;
  element3t = x[3];              /* accès par indexe */
  element3p = *(x + 3);          /*accès par pointeur*/
  printf(" x[3]= %d, *(x + 3) = %d \n", element3t,
  element3p);
}
```

Initialisation d’un tableau lors de sa déclaration

Il est possible d'affecter directement le contenu du tableau par des valeurs lors de sa déclation.

Exemples:

```
int x[3] = {1,4,121};
      /* tableau mono-dimensionnel initialisé */
int x2d[3][3]={{1,2,4},{11,12,14},{101,102,104}};
      /* tableau 2D initialisé */
```

Tableau 1D alloué dynamiquement

Souvent, la taille d'un tableau n'est connue que lors de l'exécution d'un programme. Le programme suivant alloue la taille d'un tableau à l'exécution. La fonction `malloc(nbElementsAallouer*tailleUnElementEnOctets)` effectue l'allocation mémoire du nombre d'octets désiré et rend un pointeur dont le type est converti en pointeur à un entier (int *). La fonction prédéfinie `sizeof(type)` rend la taille du type spécifié en octets.

```
#include <stdio.h>  //tableau1Ddyn.c
#include <stdlib.h>
int main() {
  int nbElements;       /* nombre d'éléments du tableau */
  int * ptrTableau;      /* pointeur au tableau qui sera alloué */
  int i;                /* indexe pour boucles */
  printf("Entrez le nombre d'éléments du tableau à allouer: \n");
  scanf ("%d", &nbElements);    /* reçoit nb eléments à allouer */
  ptrTableau = (int *) malloc(nbElements * sizeof(int));

  for( i=0; i < nbElements; i++) {
    ptrTableau[i] = i;           /* initialisation */
    printf(" ptrTableau[%d]= %d \n", i, ptrTableau[i]);
  }
   free(ptrTableau);
}
```

Exemple d'exécution

Console:

```
D:\Prog3\prog\Debug>Tableau1D
Entrez le nombre d'éléments du tableau à allouer: 18

 ptrTableau[0]= 0
 ptrTableau[1]= 1
 ptrTableau[2]= 2
 ptrTableau[3]= 3
 ptrTableau[4]= 4
 ptrTableau[5]= 5
 ptrTableau[6]= 6
 ptrTableau[7]= 7
 ptrTableau[8]= 8
 ptrTableau[9]= 9
 ptrTableau[10]= 10
 ptrTableau[11]= 11
 ptrTableau[12]= 12
 ptrTableau[13]= 13
 ptrTableau[14]= 14
 ptrTableau[15]= 15
 ptrTableau[16]= 16
 ptrTableau[17]= 17
```

Tableau 2D alloué statiquement

Un tableau de données bi-dimensionnel est représenté en langage C par:

1. **un tableau de pointeurs aux lignes du tableau de données 2D et,**
2. **par le tableau de données 2D lui-même.**

Donc, le tableau nommé tableau2D est un pointeur à un tableau de pointeurs et a donc le type int **tableau2D. Un tableau est alloué statiquement par :

```
#define NB_LIGNES 3
#define NB_COLONNES 5
int monTableau2D[NB_LIGNES][NB_COLONNES];
```

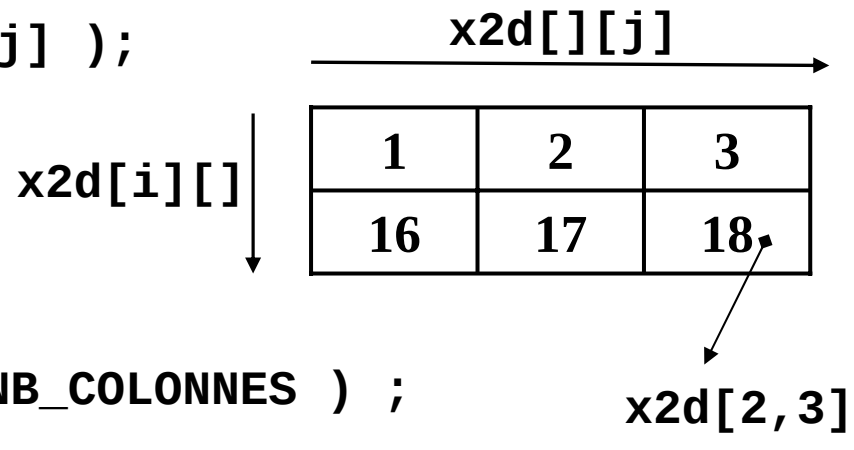
Lors de l'accès au tableau, le compilateur effectue l'opération suivante:

```
monEntier = monTableau2D[i][j] ;
monEntier =  *(*(monTableau2D + i) + j);
```

***monTableau2D* est vu comme un pointeur à un tableau de 5 entiers. *(monTableau2D+i)* pointe sur la i-ème ligne de 5 entiers. **(monTableau2D+i)* pointe sur l'élément 0 de la i-ème ligne et **(monTableau2D+i)+j* pointe sur l'élément j de la i-ème ligne du tableau. **(*(monTableau2D+i)+j)* permet donc d'accéder l'élément j de la i-ème ligne du tableau.**

Passage d'un tableau 2D déclaré statiquement comme paramètre à une procédure

```
#include <stdio.h>                                  //afficheTableau.c
#define NB_LIGNES 2
#define NB_COLONNES 3
int x2d[NB_LIGNES][NB_COLONNES] = {{1,2,3},{16,17,18}};
void afficheTableau(int tableau2D[ ][NB_COLONNES],
                    const int nblignes, const int nbcol )
{
  int i, j;
  for ( i=0; i < nblignes; i++ ) {
    for ( j=0; j < nbcol; j++ )
      printf ( "%3d", tableau2D[i][j] );
      printf ( "\n" );
   }
}
int main ( )
{
   afficheTableau( x2d, NB_LIGNES, NB_COLONNES ) ;
}
```



Affichage du tableau précédent

La fonction `displayTable` a été mise au point, afin de montrer de quelle manière sont placées les données en mémoire.

Voici le tableau présenté au slide précédent:

Console:

```
X:\prog3\Debug>afficheTableau
```

```
  1  2  3
 16 17 18
```

```c
#include <stdio.h>  // displayTable.c
//! Affiche le contenu d'un tableau en mémoire
void displayTable(void *data, int length)
{    // Caster sur un unsigned char * pour lire octets
     unsigned char *d=(unsigned char *)data;
     int i,j;

     // On veut afficher 16 octets par ligne
     for(j=0;j<(length+15)/16;j++)
     {// Pointeur en mémoire du début de la ligne courante
          printf("%8.8x ",d);
          for(i=0;i<16;i++)
          {// Affichage des valeurs en hexdécimal
            if(i+j*16<length) printf("%2.2x ",d[i]);
            else printf("   ");
            if(i==7) printf(" ");
          }
          for(i=0;i<16;i++)
          {// Affichage des valeurs en caractères
                    if(i+j*16<length) printf("%c",(d[i]>=32 && d[i]<128) ? d[i] : '.');
                    else printf(" ");
                    if(i==7) printf(" ");
          }
          printf("\n");                    // Retour à la ligne
          d+=16;                           // Inc. pointeur de 16 octets
     }
}
```

```
#define NB_LIGNES 2
#define NB_COLONNES 3

int main()
{
    short int x2d[NB_LIGNES][NB_COLONNES] = {{1,2,3},{16,17,18}};
    // Affiche le tableau
    displayTable(x2d,sizeof(x2d));
    return 0;
}
```

Console:

```
X:\prog3\Debug>displayTable
0012ff74 01 00 02 00 03 00 10 00  11 00 12 00
```

Comment vérifier l’hypothèse suivante ?

```
monEntier = monTableau2D[i][j] ;
monEntier =  *(*(monTableau2D + i) + j);
```

Vérification

```
monEntier = monTableau2D[i][j] ;
monEntier =  *(*(monTableau2D + i) + j);
```

```
int main()  // displayTable2DFixPointersMain.c
{             // same as displayTable2DFix.c & display of pointers
    short int x2d[NB_LIGNES][NB_COLONNES] = {{1,2,3},{16,17,18}};
    int i,j; short int val;
    printf("ptrTableau = %p \n", x2d);
    for (i=0;i<NB_LIGNES;i++)
          for (j=0;j<NB_COLONNES;j++)
          {
            printf("x2d[%d][%d] = %d \n", i ,  j,  x2d[i][j]);
            val= *((*(x2d + i)) + j);
             printf("ptrLigne=%p, ptrCol=%p, *((*(x2d + %d)) + %d)= %d \n",
                                                    *(x2d + i),(*(x2d + i)) + j, i,j,x2d[i][j]);
          }
    // Affiche le tableau
    displayTable(x2d,sizeof(x2d));

    return 0;
}
```

Console

```
ptrTableau = 0012FF74
x2d[0][0] = 1
ptrLigne=0012FF74, ptrCol=0012FF74, *(*(x2d + 0) + 0)= 1
x2d[0][1] = 2
ptrLigne=0012FF74, ptrCol=0012FF76, *(*(x2d + 0) + 1)= 2
x2d[0][2] = 3
ptrLigne=0012FF74, ptrCol=0012FF78, *(*(x2d + 0) + 2)= 3
x2d[1][0] = 16
ptrLigne=0012FF7A, ptrCol=0012FF7A, *(*(x2d + 1) + 0)= 16
x2d[1][1] = 17
ptrLigne=0012FF7A, ptrCol=0012FF7C, *(*(x2d + 1) + 1)= 17
x2d[1][2] = 18
ptrLigne=0012FF7A, ptrCol=0012FF7E, *(*(x2d + 1) + 2)= 18

0012ff74 01 00 02 00 03 00 10 00  11 00 12 00          ........ ....
```

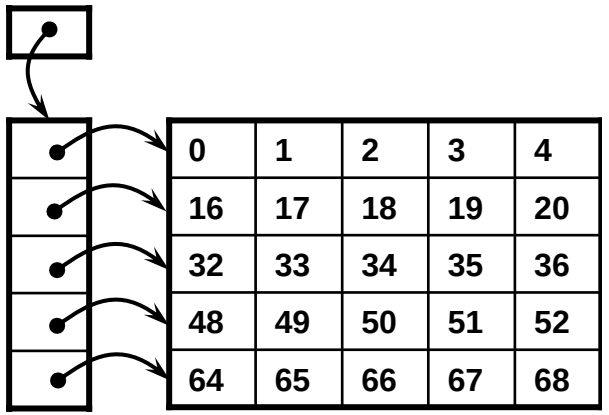
Tableau 2D alloué dynamiquement

```c
#include <stdio.h>                          //tableau2Ddyn.c
#include <stdlib.h>
int main(void) {
  int i, j, nbColonnes, nbLignes;
  int **ptrTableau;/*ptrTableau est un pointeur à un tableau de pointeur*/
  printf(" Entrez le nombre de colonnes du tableau à allouer: \n");
  scanf ("%d", &nbColonnes);
  printf(" Entrez le nombre de lignes du tableau à allouer: \n");
  scanf ("%d", &nbLignes);
       /* allocation de la mémoire au tableau de pointeurs aux lignes */
  ptrTableau = (int **) malloc(nbLignes * sizeof(int *));
       /* allocation de la memoire au tableau bidimensionnel */
  *ptrTableau = (int *) malloc(nbColonnes * nbLignes * sizeof(int));

  for(i=0; i < nbLignes; i++)
     /*assignation à chaque pointeur de la ligne sur laquelle il pointe*/
    ptrTableau[i] = ptrTableau[0] + nbColonnes * sizeof(int) * i;
  for(i=0; i < nbLignes; i++){
    for(j=0; j < nbColonnes; j++) ptrTableau[i][j] = 16*i+j;
   }    /* initialisation des éléments du tableau */
  for (i=0; i < nbLignes; i++)       {
    for (j=0; j < nbColonnes; j++)
       /*affichage d'éléments référencés (a)par index (b)par pointeur*/
      printf("%4d %2d ",ptrTableau[i][j], *(*(ptrTableau+i)+j));
      printf("\n");           }
}
```

Exemple d'exécution

Console:

```
D:\Prog3\prog\Debug> Tableau2D
Entrez le nombre de colonnes du tableau à allouer:
5
Entrez le nombre de lignes du tableau à allouer:
5
 0  0    1  1    2  2    3  3    4  4
16 16   17 17   18 18   19 19   20 20
32 32   33 33   34 34   35 35   36 36
48 48   49 49   50 50   51 51   52 52
64 64   65 65   66 66   67 67   68 68

002f07a8|f0 07 2f 00 40 08 2f 00  90 08 2f 00 e0 08 2f 00
002f07b8|30 09 2f 00
```



- **Nous n'accédons pas directement aux valeurs, mais aux positions du tableau à travers les pointeurs de lignes.**

```
*(Tableau2D[noLigne] + noElem)
```

Passage d'un tableau 2D dynamique comme paramètre à une procédure (2eme exemple)

```
#include <stdio.h> //afficheTableau2Ddyn.c
#include <stdlib.h>
void afficheTableau( int **tableau2D, const int nblignes,
                                      const int nbcol)
{
  int i, j;
  for(i=0; i < nblignes; i++) {
    for(j=0; j < nbcol; j++)
      printf("%3d",tableau2D[i][j]);
  printf("\n");
  }
}
```

Suite:

```
int main(void) {
  int i, j, nbColonnes, nbLignes;
  int **ptrTableau;
   /*ptrTableau est un pointeur à un tableau de pointeur*/
  printf("Entrez le nombre de colonnes du tableau à allouer: \n");
  scanf ("%d", &nbColonnes);
  printf("Entrez le nombre de lignes du tableau à allouer: \n");
  scanf ("%d", &nbLignes);
  /* allocation de la mémoire au tableau de pointeurs aux lignes*/
  ptrTableau = (int **) malloc(nbLignes * sizeof(int *));
  /* allocation de la memoire au tableau bidimensionnel */
  *ptrTableau=(int *) malloc(nbColonnes * nbLignes * sizeof(int));

//assignation à chaque pointeur de la ligne sur laquelle il pointe
   for(i=0; i < nbLignes; i++)
   ptrTableau[i] = ptrTableau[0] + nbColonnes * sizeof(int) * i;
  for(i=0; i < nbLignes; i++){
    for(j=0; j < nbColonnes; j++) /* initialisation des éléments */
      ptrTableau[i][j] = 16*i+j;
  }
  afficheTableau(ptrTableau, nbLignes, nbColonnes);
  displayTable(ptrTableau, nbLignes * sizeof(int*));
}
```

Exécution et affichage

Console:

```
X:\prog3\Debug>afficheTableau2
Entrez le nombre de colonnes du tableau a allouer:
5
Entrez le nombre de lignes du tableau a allouer:
5
  0  1  2  3  4
 16 17 18 19 20
 32 33 34 35 36
 48 49 50 51 52
 64 65 66 67 68

002f07a8|f0 07 2f 00 40 08 2f 00  90 08 2f 00 e0 08 2f 00
002f07b8|30 09 2f 00
```

- **Nous pouvons voir ici que nous n'accédons pas directement aux valeurs, mais aux pointeurs de ligne du tableau.**

Affichage et architecture mémoire

```
Console:
002f07a8 f0 07 2f 00 40 08 2f 00  90 08 2f 00 e0 08 2f 00
002f07b8 30 09 2f 00
002f07f0 00 00 00 00 01 00 00 00  02 00 00 00 03 00 00 00
002f0800 04 00 00 00
002f0840 10 00 00 00 11 00 00 00  12 00 00 00 13 00 00 00
002f0850 14 00 00 00
002f0890 20 00 00 00 21 00 00 00  22 00 00 00 23 00 00 00
002f08a0 24 00 00 00
002f08e0 30 00 00 00 31 00 00 00  32 00 00 00 33 00 00 00
002f08f0 34 00 00 00
002f0930 40 00 00 00 41 00 00 00  42 00 00 00 43 00 00 00
002f0940 44 00 00 00
```

- **On retrouve bien les valeurs du tableau aux adresses indiquées.**

Structures

Les structures sont des types définis par le programmeur. Elles permettent de déclarer des variables qui comportent plusieurs champs, chaque champ pouvant être d'un type différent. Par exemple la structure qui suit décrit le type adresseType.

```
struct adresseType {
  char nom[20];
  char rue[20];
  char ville[20];
  int  numeroPostal;
};
```

Une fois le type déclaré, celui-ci peut être réutilisé pour déclarer des variables, par exemple la variable `adresseMonPere`.

```
struct adresseType adresseMonPere;
```

Accéder à un champs d'une structure

Pour accéder à un champs qui se trouve au sein d'une variable de type `struct`, on spécifie nomVariable.champs, par exemple :

```
int monNumeroPostal = 1066;
monAdresse.numeroPostal = monNumeroPostal;
```

Souvent, on alloue dynamiquement une structure, puis on accède à ses champs à travers le pointeur à la structure en utilisant le signe "->“

```
struct adresseType *ptrAdressePere;
  /* déclaration du pointeur à la structure */
ptrAdressePere= malloc(sizeof(addresseType));
  /*allocation de la mémoire */
ptrAdressePere->numéroPostal = 1066;
  /* Accès à un champ de la structure */
(*ptrAdressePere).numéroPostal = 1066;
```

Définition de structures

On définit à l'aide de `typedef` un nouveau type qui est une structure :

```
typedef struct {
     /* déclaration d'un nouveau type */
  char nom[20];
     /* chaîne de caractères */
  float masse;
  float taille;
  /* nombres réels virgules flottantes */
} personneT;
     /* nom du nouveau type définit */
```

Définition de structures

Déclaration d'une variable de type personneT:

```
personneT jean;
jean.nom = "Jean";
           /*assignation d'une chaîne de caractères*/
jean.masse = 80.5;      /* assignation d'une valeur */
jean.taille = 185.4;
```

Ou

```
personneT jean = {"Jean", 80.5, 186.4};
personneT jacques = {"Jacques", 75.2, 175.8};
```

Une variable de type struct peut être affectée à une autre de même type -> tous les champs seront copiés:

```
personneT cloneJean = jean;
```

Nom	Masse	Taille
Jean	80.5	185.4
Jacques	75.2	175.8

Tableau de structures:

```
personneT maClasse[10];
maClasse[0] = jean; maClasse[1] = jacques;
```

Union

L'union représente un type dont le contenu, c'est à dire les champs peuvent être vus de différentes manière. Dans l'exemple qui suit, on cherche à manipuler un ensemble de deux octets soit sous forme d'un entier 16 bits (type short int), soit sous forme d'une suite de deux caractères.

```
#include <stdio.h>  /* union.c */
union modeleT {                    /* définition du type */
  short int i;                     /* entier 16 bits */
  char ch[2];                      /* chaîne de deux caractères */
};
union modeleT maVariable;          /* variable globale */

void affiche (union modeleT mot16bits){
  printf(" 1er car: %c, 2ème car: %c\n",//affiche 1er & 2ème car
              mot16bits.ch[0], mot16bits.ch[1] );
  printf("entier i sous forme hexadécimale: %X\n",mot16bits.i);
}
int main() {
  maVariable.i = 0x4142;           /* initialise l'entier 16 bits
   par les codes hexadécimaux des caractères "A" et "B" */
  affiche(maVariable);
}  //Qu'est-ce qui est affiché ?
```

Enumérations

Le type énumération permet d'associer à une variable une parmi plusieurs valeurs constantes définies par l'utilisateur. Le programme suivant définit le type `enum piece` dont une variable de ce type peut prendre une des valeurs `{penny, quarter, dollar}`.

```
#include <stdio.h>
int main() {
  enum piece {penny, quarter, dollar};
  enum piece argent;
  argent = quarter;
  switch(argent){
     case penny:    printf("penny"); break;
     case quarter:  printf("quarter"); break;
     case dollar:   printf("dollar");
  }
}
```

Lecture/écriture de fichiers

Ouverture, lecture, écriture et fermeture de fichiers

Tout fichier doit être préalablement ouvert avant de pouvoir effectuer des opérations de lecture ou d'écriture. Il est possible d'ouvrir un fichier dans les modes suivants:

"r"	**en lecture ,**
"w"	**en écriture (nouveau fichier),**
"a"	**adjonction (append),**
"r+"	**en lecture/écriture (fichier existe déjà),**
"w+"	**en écriture/lecture ("w+") (nouveau fichier),**
"rb"	**en lecture binaire,**
"wb"	**en écriture binaire,**
"ab"	**en mode adjonction binaire (binary append),**
"r+b"	**en mode lecture/écriture binaire,**
"w+b"	**en mode écriture/lecture (création de fichier).**

rb, wb, ab, r+b : ne transforme pas les retours de ligne

Attention: Différences entre Windows et UNIX

- **Sous UNIX, la fin de ligne est le \n (LF)**
- **Sous Windows, la fin de ligne est le \r\n (CR LF)**

Lecture/écriture de fichiers (2)

La fonction d'ouverture a le format suivant :

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

La variable `const char *filename` est un pointeur à une chaîne de caractères. Le type en retour `FILE *` est un pointeur vers un descripteur de fichier (ou de flux). Si le pointeur vers le descripteur de fichier est zéro, c'est-à-dire NULL, une erreur a eu lieu.

Lecture/écriture de fichiers (3)

Une fois le fichier ouvert, l'opération de lecture ou écriture peut débuter. Par défaut, un pointeur à la position courante dans le fichier est maintenu (pointeur à un octet). Ce pointeur est mis à jour à chaque lecture ou écriture. Pour lire, respectivement écrire un octet, il est possible d'utiliser les fonctions :

```
int fgetc(FILE *ptrFile0)
  /* lit un octet et le place dans un
  entier */
int fputc(int ch, FILE *ptrFile1);
  /* écrit un octet sur un fichier */
```

L'octet lu ou écrit se trouve sur l'octet de poids faible de l'entier. `fputc` retourne un `EOF` lorsque la fin du fichier est atteinte. `fputc` rend le caractère écrit sur le fichier; en cas d'erreur, la valeur `EOF` est retournée.

Lecture/écriture de fichiers (4)

Il est possible de vérifier si l'on se trouve à la fin du fichier par l'appel :

```
int feof(FILE *ptrFile);
```

qui rend une valeur non-nulle (true) si on se trouve à la fin du fichier et une valeur zéro (false) si on ne se trouve pas à la fin du fichier.

Après l'ouverture et l'opérations de lecture ou d'écriture sur un fichier, il est nécessaire de le fermer par l'appel :

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

qui en cas de succès de la fermeture rend un zéro et en cas d'erreur rend EOF.

Exemple lecture/écriture de fichier

```
#include <stdio.h>     // io.c : ouverture d'un fichier en lecture
#include <stdlib.h>    // création d'un fichier en écriture
int main () {          // lecture et écriture de caractères
  FILE *fichEntree, *fichSortie;
  char nomFichEntree[24], nomFichSortie[24];
  int ch;
  printf("Entrez le nom du fichier d'entrée\n");
  scanf("%s", nomFichEntree);
  printf("Entrez le nom du fichier de sortie\n");
  scanf("%s", nomFichSortie);
  if((fichEntree = fopen(nomFichEntree, "rb")) == NULL){
    printf("Impossible d'ouvrir le fichier source\n");
    exit(1);
  }
  if((fichSortie = fopen(nomFichSortie, "wb")) == NULL){
    printf("Impossible de créer le fichier de destination\n");
    exit(1);
  }
    while(!feof(fichEntree)){          // copie le fichier source dans le
    ch = getc (fichEntree);            // fichier destination
    if(!feof(fichEntree)) putc(ch,fichSortie);
  }
  fclose(fichEntree);
  fclose(fichSortie);
}
```

Lecture de fichier chaîne par chaîne

```
#include <stdio.h>     // ioString.c: lecture d'un fichier mot par mot
#include <stdlib.h>    // chaque mot est terminé par CR LF ou LF
#define LENGTH_MAX 64
FILE * fptr;
char buf[LENGTH_MAX ]; // Assume no word is longer than LENGTH_MAX

int main(int argc, char *argv[]){
  if(argc!=2) {
    printf("Please specify input file name\n");
    return 1;
  }

       // le nom du fichier est passé sur la ligne de commande
   fptr=fopen(argv[1],"r"); //  Open file
  if(fptr==NULL) {
       // File could not be opened, print error message
   printf("Could not open file %s \n",argv[1]);
   return 1;
  }

       // Lire mots du fichier (terminés par espace ou line end)
  while(fscanf(fptr,"%s",buf)!=EOF) printf(" mot: %s\n",buf);
  return 0;
}
```

Ecriture/lecture de chaînes de caractères

En plus des fonctions de lecture et d'écriture d'octets, il est possible de lire ou d'écrire des chaînes de caractères à l'aides des fonctions :

```
char *fgets(char *s, int length, FILE *ptrFile0);
int  fputs(const char *s, FILE *ptrFile1);
```

`fgets` essaie de lire dans `*s` une chaîne de caractères qui compte au maximum `length` caractères, terminée soit par une fin de ligne `\n`, soit par une fin de fichier `EOF`. La chaîne résultante est terminée par le caractère zéro `\0`.
`fputs` écrit la chaîne de caractères spécifiée par `*s` et terminée par le caractère `\0` dans le fichier et rend zéro s'il n'y a pas d'erreurs.

A la place d'utiliser `fputs`, il est possible d'utiliser `fprintf` de manière similaire à `printf` :

```
int fprintf(FILE *ptrFichier, const char
            *SpécificateurFormat, arg1, arg2, ...);
```

Lecture de chaîne de caractères par `fgets`

```
#include <stdio.h>     // fgets.cpp : lecture d'un fichier mot par mot
#include <stdlib.h>    // chaque mot est terminé par CR LF ou LF
#define LENGTH_MAX 64

int main(int argc, char *argv[]){
  if(argc != 2) {
    printf("Please specify input file name\n");
    return 1;
  }

        // le nom du fichier  passé sur ligne de commande
  FILE * fptr;
  fptr = fopen(argv[1],"r"); //  Open file
  if(fptr==NULL) {
        // le fichier ne peut être ouvert
   printf("Could not open file %s \n",argv[1]);
   return 1;
  }

        // Lire mots du fichier (terminés par CR LF)
  char buf[LENGTH_MAX]; // No word is longer than LENGTH_MAX
  while(fgets(buf, LENGTH_MAX, fptr) != NULL)
        printf(" mot: %s",buf);
  return 0;
}
```

Exemple de lecture/écriture de chaîne de caractères

```
#include <stdio.h>   // wr.c Ecriture d'une chaîne de caractère
#include <stdlib.h>  // dans un nouveau fichier
#include <string.h>
int main(void) {
   char ligne[80];
    FILE *ptrFichier;

    if((ptrFichier = fopen("TEST.txt","w")) == NULL){
            printf("Impossible d'ouvrir le fichier\n");
            exit(1);
    }
    do{
            printf("entrez une ligne de texte:\n");
            gets(ligne);
            fprintf(ptrFichier, "%s\n",ligne);
            //écriture de la chaîne dans le fichier
    } while(strcmp(ligne, ""));
    // tant que la ligne n'est pas vide (si vide, rend zero)
    fclose(ptrFichier);
}
```

Ecriture/lecture d'un ensemble d'éléments

Il est possible de lire dans un fichier un ensemble de `nbElem` éléments, chacun de taille `tailleElem` et de les placer dans `*tampon` à l'aide de la fonction :

```
size_t fread(void *tampon, size_t tailleElem,
             size_t nbElem, FILE *ptrFichier0);
```

Similairement, il est possible d'écrire dans un fichier `ptrFichier1` un ensemble de `nbElem` éléments, chacun de taille `tailleElem` qui se trouvent dans le tampon pointé par `*tampon` :

```
size_t fwrite(const void *tampon,
              size_t tailleElem,
              size_t nbElem, FILE *ptrFichier1);
```

```c
// freadfwrite.c: ouvre fichier, écrit 26 caracteres et relit le fichier
#include <stdio.h>   //  VC++, RDH
#define NB_CHARS 26
void main( void )
{  FILE *descrFich;
   char tabCar0[NB_CHARS+1]; char tabCar1[NB_CHARS+1];
   int  i, nbCarLus, nbCarEcrits;
   // Ouvre fichier fwrite.txt en mode texte
   if( (descrFich = fopen( "fichier.txt", "w+t" )) != NULL )
   {  // initialiser tableau de caractères
      for ( i = 0; i < NB_CHARS; i++ ) tabCar0[i] = (char)('a' + i);
      tabCar0[NB_CHARS]=0;
      // Ecrit NB_CHARS caractères dans fichier
      nbCarEcrits = fwrite( tabCar0, sizeof( char ), NB_CHARS+1, descrFich );
      printf( "Ecrit %d caracteres\n", nbCarEcrits );
      fclose( descrFich );
   }
   else printf( "Probleme ouverture de fichier en ecriture\n" );
   // ouverture du fichier en lecture, mode texte
   if( (descrFich = fopen( "fichier.txt", "r+t" )) != NULL )
   {  // essaie de lire les caractères
      nbCarLus = fread( tabCar1, sizeof( char ), NB_CHARS+1, descrFich );
      printf( "Nombre de caracteres lus: \t%d\n", nbCarLus );
      printf( "Contenu du tableau lu:    \t%s\n", tabCar1 );
      fclose( descrFich );
   }
   else
      printf( "Probleme ouverture de fichier en lecture\n" );
}
```

Repositionnement du pointeur dans un fichier

Il peut être nécessaire de repositionner le pointeur à une position donnée du fichier. La fonction `fseek` en mode SEEK_SET positionne la valeur courante du pointeur à la position offset depuis le début du fichier.
Les lectures et écritures suivantes sont effectuées à partir de la position pointée.

```
int fseek(FILE *ptrFichier,
          long int offset, SEEK_SET);
```

Sans erreur, `fseek` revient avec la valeur zéro.

Exemple: **fseek**

```
#include <stdio.h>   //fseek.c
#include <stdlib.h>
int main() {
  FILE *fp;
  int noOctet;
  char nomFichier[24];
  printf("Entrez le nom du fichier et le no d'octet à
   inspecter\n");
  scanf("%s\n%d", nomFichier, &noOctet);
  if((fp = fopen(nomFichier, "r")) == NULL){
    printf("\nImpossible d'ouvrir le fichier");
    exit(1);
  }
  if(fseek(fp,noOctet, SEEK_SET)){
    printf("erreur de seek\n");
    exit(1);
  }
  printf("Octet à position %d est %c\n",noOctet,fgetc(fp));
}
```

Flux d'entrée-sortie standard

Toute opération de lecture sur fichier peut également s'effectuer sur l'entrée standard, c'est-à-dire sur le terminal ou la console (comme nom de flux indiquer `stdin`). Toute opération d'écriture peut s'effectuer sur la sortie standard, c'est-à-dire sur le terminal ou la console (comme nom de flux indiquer `stdout`).

Exemple:

```
fputs("Bonjour", stdout);
```

Pointeur à une fonction

Il peut être utile de connaître l'adresse d'une fonction, par exemple pour la programmation d'une routine d'interruption. Pour déclarer un type pointeur à une fonction qui rend un entier :

```
int (*ptrFunc)();
```

et pour ensuite associer à ce pointeur une fonction, on définit d'abord la fonction :

```
int maFonction()
{      /* mes instructions */      }
```

Puis, on initialise le pointeur dont la valeur est désormais connue :

```
ptrFunc = maFonction;
printf("valeur du pointeur à la fonction %d",
                                    (int) ptrFunc);
```

Pour appeler la fonction dont on connaît le pointeur, effectuer :

```
int resultat;
resultat = (*ptrFunc)();
```

Exemple de pointeur à une fonction

Le programme suivant définit un type pointeur à une fonction (main), on lui associe une fonction librairie (provenant de string.h) et le passe comme paramètre à une fonction. Ainsi, à une autre occasion, on pourrait passer une autre fonction comme paramètre.

```
#include <stdio.h>  //ptrFunction.c
#include <string.h>              /* importe la fonction strcmp */
void test(char *a,char *b,int (*ptr)(const char *, const char *))
{                                /* ptr pointe sur une fonction */
  printf("Test d'égalite:\t");
  if (!(*ptr)(a, b)) printf("égal \n");
                          /* a et b égaux si ptr à fonc rend zéro*/
    else printf("pas égal \n");
}
int main() {
  char chaine1[80], chaine2[80];
  int  (*monPointeur) (const char *, const char *);
                          // monPointeur pointe sur une fonction
  monPointeur = strcmp;//strcmp rend 0 si les chaines sont égales
  printf("entrez deux chaines de caractères successives\n");
  scanf("%s\n%s", chaine1, chaine2);
  test(chaine1, chaine2, monPointeur);
                /* passe le ptr à la fonction comme paramètre */
}
```

Gestion d'opérateurs

```
#include <string.h> // tableOpFunction.c TB, RDH
#include <stdio.h>
#include <stdlib.h>
#define NB_OF_OPERATIONS 4
/* Type definition of an operation handler */
typedef double(*opHandler)(double, double);
/* Definitions of the possible operations */
double doAddition(double left, double right)
{  return left + right;
}
double doMultiplication(double left, double right)
{  return left * right;
}
double doDivision(double left, double right)
{  return left / right;
}
double doSubtraction(double left, double right)
{  return left - right;
}
```

```
// List of the operation handlers : pointers to functions
void * operationHandlers[NB_OF_OPERATIONS][2] =
   {{"+", doAddition }, {"*", doMultiplication},
   {"/", doDivision  }, {"-", doSubtraction  }};

typedef struct {
  double left; double right; char op[2];   int error;
} instructionT;

// Get the three parts of the instruction: leftNb Op rightNb
instructionT getNextInstr()
{ char line[256]; char * token; instructionT instr;
  fgets(line, 256, stdin);  token = strtok(line, " \t\n");
  if (token == NULL) {instr.error = -1;return instr; }
  instr.left = atof(token);

  token = strtok(NULL, " \t\n");
  if (token == NULL) {instr.error = -1; return instr;}
  strncpy(instr.op, token, 1); instr.op[1] = '\0';

  token = strtok(NULL, " \t\n");
  if (token == NULL)  {instr.error = -1; return instr;}
  instr.right = atof(token); instr.error = 0;
  return instr;
}
```

```
int main(int argc, char *argv[])
{
  int i;
  instructionT next;
  printf("Enter instruction in format: nb <sp> operation <sp> nb : ");
  next= getNextInstr();
  if (next.error !=0) { printf("error no : %d\n", next.error); return 1;}
  while(next.error == 0)
  {
    for (i = 0; i < NB_OF_OPERATIONS; i++)
    {
      if (strcmp(next.op, operationHandlers[i][0]) == 0)
      {
        double result;
        opHandler handler = (opHandler)operationHandlers[i][1];
        result = handler(next.left, next.right);
        printf("\nResult: %lf\n\n", result);
        break;
      }
    }
    if (i == NB_OF_OPERATIONS)
      printf("\nUnknown operation: %s\n\n", next.op);
    printf("Next instruction: ");
    next = getNextInstr();
   if (next.error !=0) { printf("error no : %d\n", next.error); return 1;
   }
  } return 0;
}
```

Fonctions de la librairie String.h

Attention: les débordements de buffer ne sont pas testés ➔ erreurs fatales!

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

Copie la chaîne `src` dans la chaîne `dest`. Retourne `dest`.

```
char* strncpy(char* dest, char const* src, size_t n);
```

Copie les n premiers caractères de `src` dans `dest`. Retourne `dest`.

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

Ajoute la chaîne `src` à la fin de la chaîne `dest`. Retourne `dest`.

```
char* strncat(char* dest, char const* src, size_t n);
```

Ajoute au plus n caractères de `src` à la fin de `dest`. Retourne `dest`.

Source: J.C. Chappelier

Fonctions de la librairie String.h (2)

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

Compare (ordre alphabétique) les chaînes `s1` et `s2`. Rend 0 si les deux chaînes sont identiques, un nombre négatif si `s1 < s2`, et un nombre positif si `s1 > s2`.

```
int strncmp(char const* s1, char const* s2, size_t n);
```

Comme `strcmp`, mais ne compare au plus que les n premiers caractères de chacune des chaînes.

```
size_t strlen(char const* s);
```

Retourne le nombre de caractères dans `s` (sans le caractère nul de la fin).

```
char* strchr(char const* s, char c);
```

Retourne un pointeur sur la première occurrence de `c` dans `s`, ou `NULL` si `c` n'est pas dans `s`.

```
char* strstr (char const* s1, char const* s2);
```

Retourne un pointeur sur la première occurrence de `s2` dans `s1` (ou `NULL` si `s2` n'est pas incluse dans `s1`).

Source: J.C. Chappelier

Compilation séparée

Un programme complexe comprend de nombreux modules qui peuvent être compilés séparément. Au cas où l'application comprend 2 ou plusieurs fichiers, les variables qui se trouvent dans un autre fichier sont référencées par le mot `extern`.

fileMain.c

```
#include "fileAux.h"
int x, y;
char ch;
void func1( )
{
  x = 123;
}
int main(void)
{ func1();
  func23();
  func22();
  printf("x = %i \n",x);
  exit (0);
}
```

fileAux.c

```
extern int x, y;
extern char ch;
void func22(void)
{
  x=y/10;
}
void func23( )
{
y = 10;
}
```

fileAux.h

```
void func22(void);
void func23(void);
```

Compilation séparée (2)

Pour compiler:

```
gcc –c fileAux.c –o fileAux.o
```

crée le code objet fileAux.o.

```
gcc –c fileMain.c –o fileMain.o
```

crée le code objet fileMain.o.

```
gcc fileMain -o fileAux.o –o fileExec
```

crée le fichier exécutable à partir des 2 fichiers objets.

Ceci est équivalent à la ligne de commande suivante:

```
gcc fileMain.c fileAux.c –o fileExec
```

Makefile

Source: J.C. Chappelier

Un Makefile a une structure très simple : il est constitué d'un ensemble de règles décrivant les différents modules à créer et de quoi ils dépendent (« liste de dépendances »).
But: ne compiler que les programmes qui ne sont pas à jour

Une règle s'écrit : `but: liste de dépendances`

Exemple :
```
fileExec: fileMain.o fileAux.o
```

La règle spécifique `all` **donne la liste des exécutables que l'on veut créer:**
```
all: fileExec
```

Si on a des librairies systèmes à utiliser, il faut les ajouter dans la variable `LDLIBS` au début du Makefile :
```
LDLIBS = -lm
```
pour les librairies mathématiques

Exemple de « Makefile » (1)

```
LDLIBS = -lm
all: fileExec
fileExec: fileMain.o fileAux.o
   gcc -o $@ $^ $(LDLIBS)
```

- `$@` **full target name (fileExec)**
- `$*` **target name without extension**
- `$^` **following part of string**
- `$(LDLIBS)` **–lm**

Source: J.C. Chappelier

Exemple de « Makefile » (2)

```
LDLIBS = -lm
all: fileExec
fileExec: fileMain.o fileAux.o
	gcc -o $@ $^ $(LDLIBS)
```

Remarques:

1. **Il n'est pas nécessaire de mettre des règles simples indiquant la dépendance entre le .c et le .o correspondant (comme par exemple fileMain.o: fileMain.c). Ce sont des règles implicites.**
2. **Si on veut ajouter des options au compilateur, on peut le faire avec la variable `CPPFLAGS` :**

```
CPPFLAGS = -ansi -q
```


Source: J.C. Chappelier