

Langage C

introduction générale

Prof. A.SABOUR

Objectifs du cours ...

Objectif :

- Ce cours a pour objectif de procurer une connaissance moderne de la programmation afin qu'un étudiant puisse résoudre des problèmes.
- Le langage de programmation utilisé est le C ANSI 89.
- De façon plus spécifique, ce cours devra permettre à l'étudiant de :
 - acquérir les notions de programmation de base;
 - acquérir une connaissance du langage C ;
 - Être capable de traduire un algorithme en un programme C
 - Comprendre les différentes **constructions** de la programmation en **C**

Historique

- 1972 : Dennis Richie & Ken Thompson (Bell Labs) pour développer le **système UNIX**
- 1980 : C devient **populaire** \Rightarrow plusieurs **compilateurs**
- 1983 : **ANSI** (American National Standards Institute) **normalise** le langage
- 1989 : fin de la **normalisation** qui donne la **norme ANSI C**

Généralités

- Langage successeur du **langage B** dans les années 60, premières normes en 1978 puis norme ANSI en 1989 et ISO en 1990.
- C n'est lié à **aucune** architecture **particulière** ;
- C est un langage type qui fournit **toutes** les instructions nécessaires à la programmation **structurée** ;
- C est un langage compilé. (et non **interprété**)
- Usage des **pointeurs**, **récurtivité**
- Langage **polyvalent** permettant le développement de **systèmes d'exploitation**, de programmes **applicatifs scientifiques** et de **gestion**.
- Langage **évolué** qui permet néanmoins d'effectuer des opérations de **bas niveau**.
- **Portabilité** (en **respectant** la **norme** !) due à l'emploi de bibliothèques dans lesquelles sont **reléguées** les fonctionnalités liées à la machine.
- Grande **efficacité** et **puissance**.

C est un langage de bas niveau

- Il n'est pas rare d'entendre dire que C est un **assembleur de haut niveau** i.e. un assembleur type qui offre des structures de contrôle élaborées et qui est –**relativement**– portable (et **porte**) sur l'ensemble des architectures.
- Ce langage est pensé comme un assembleur portable : son pouvoir d'expression est une **projection** des **fonctions élémentaires** d'un **microprocesseur idéalisé** et suffisamment simple pour être une abstraction des architectures réelles.

La fonction principale

- La fonction « **main** » contient le **programme principal**
- Le programme exécutable binaire commence par exécuter les instructions de ce **programme principal**
- Sans la fonction **main**, il est **impossible** de **générer** un programme **exécutable**

La fonction principale

TYPE de la valeur de
retour

"**main**" : Cela signifie "principale",
ses instructions sont exécutées.

int main(void)

}

debut /* corps du programme*/
declaration des Cstes et Var ;

instruction1 ;

instruction2 ;

....

}

fin

void main(void): La fonction main ne
prend aucun paramètre et ne retourne pas
de valeur.

int main(void): La fonction main
retourne une valeur **entière** à l'aide de
l'instruction return (o si pas d'erreur).

int main(int argc, char *argv[]): On
obtient alors des programmes auxquels
on peut adresser des arguments au
moment où on lance le programme.

Entre accolades "{" et "}" on
mettra la succession
d'actions à réaliser.(Bloc)

Structure d'un programme

- Un **programme** est composé de plusieurs **fonctions** qui **échangent** et **modifient** des **variables**
- Un **processus** est **l'abstraction** d'un programme **exécute** par la machine.

programme sur le disque

Magic Number
entête
Code
données initialisées
table des symboles

Séparé en bloc

processus en mémoire

Pile d'exécution
↓
↑
tas (malloc)
données non initialisées
données initialisées
code

Séparé en page

Les composantes élémentaires du C

- Un programme en C est constitué de 6 composantes élémentaires :
 - identificateurs
 - mots-clefs
 - constantes
 - chaînes de caractères
 - opérateurs
 - signes de ponctuation
 - + les commentaires

Identificateurs

- Un **identificateur** peut désigner :
 - Nom de **variable** ou fonction
 - type défini par *typedef*, *struct*, *union* ou *enum*,
 - étiquette
- un **identificateur** est une **suite** de **caractères** :
 - **lettres**, chiffres, « blanc souligné » (`_`)
- **Premier caractère** n'est **jamais** un **chiffre**
- minuscules et majuscules sont **différenciées**
- **Longueur** ≤ 31

un nom associe a de l'espace mémoire

Identificateurs

Exemples :

abc, Abc, ABC sont des identificateurs valides et tous différents.

Identificateurs valides :

xx y1 somme_5 _position

Noms surface fin_de_fichier VECTEUR

Identificateurs invalides :

3eme commence par un chiffre

x#y caractère non autorisé (#)

no-commande caractère non autorisé (-)

taux change caractère non autorisé (espace)

Et pour les types !!!

Types de variables manipulées en C

- Toutes les **variables** doivent être **explicitement** typées (pas de déclaration implicite)
- Il y a globalement trois types de variables :
 - Les entiers : **int, short int, long int**
 - Les réels : **float, double, long double**
 - Les caractères : **char**
 - Rien ... : **void**
- exemples : `short int mon_salaire;`
`double cheese;`
`char avoile;`

Types de base

4 types de base, les autres types seront **dérivés** de ceux-ci.

Type	Signification	Exemples de valeur	Codage en mémoire	Peut être
char	Caractère unique	'a' 'A' 'z' 'Z' '\n' 'a' 'A' 'z' 'Z' '\n' Varie de -128 à 127	1 octet	signed, unsigned
int	Nombre entier	0 1 -1 4589 32000 -231 à 231 +1	2 ou 4 octets	Short, long, signed, Unsigned,
float	Nombre réel simple	0.0 1.0 3.14 5.32 -1.23	4 octets	
double	Nombre réel double précision	0.0 1.0E-10 1.0 - 1.34567896	8 octets	long

Mots-clefs

- Réservés pour le langage lui-même et **ne** peuvent être **utilisés** comme identificateur, 32 mots-clefs :
 - const, double, int, float, else, if, etc.

Commentaires

Débute par */** et se termine par **/*

/ Ceci est un commentaire */*

// Ceci est un commentaire

Un identificateur ne peut pas être un mot réservé du langage :

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

double, int, long,
char, const, float,
short, unsigned,
signed, void, sizeof

11

break, else, switch,
case, return,
continue, for,
default, goto, do, if,
while

12

struct, enum,
typedef, union

4

auto, register,
extern, volatile,
static

5

Les mots réservés du langage C doivent être écrits en minuscules.

Un identificateur ne peut pas être un mot réservé du langage :

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

Les 32 mots-clefs de l'ANSI C

- ▶ les spécificateurs de type :

char	double	enum	float	int	long
short	signed	struct	union	unsigned	void

- ▶ les qualificateurs de type : const volatile

- ▶ les instructions de contrôle :

break	case	continue	default	do	else
for	goto	if	switch	while	

- ▶ spécificateurs de stockage :

auto	register	static	extern	typedef
------	----------	--------	--------	---------

- ▶ autres : return sizeof

Les 40 opérateurs de l'ansi C

- les opérateurs

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

Structure d'un programme C

- Une **expression** est une **suite** de composants élémentaires **syntactiquement correcte**, par exemple :

`x = 0`

`(i >= 0) && (i < 10) && (p != 0)`

- Une **instruction** est une **expression** suivie d'un **point-virgule** (fin de l'instruction)

Par exemple, l'affectation `foo = 2` provoque :

- l'effet latéral : l'entier 2 est affecté à la variable `foo` ;
- et retourne la valeur qui vient d'être affectée.

`if (x=1) printf("\n tropical \n");`

`If (x=0) printf("\n tropical \n");`

Structure d'un programme C

- Plusieurs **instructions** peuvent être rassemblées par des **accolades** { } et forme un **bloc**, par exemple :

```
if (x != 0)
{
    z = y / x;
    t = y % x;
}
```

Qu'est-ce qu'un bloc d'instructions ?

- Un bloc débute par une **accolade ouvrante** et se termine par une **accolade fermante**
- Il contient des déclarations de **variables internes** au bloc et des **instructions**
- Les lignes d'instructions se terminent par des **points virgules**.

Structure d'un programme C

- Une **instruction** composée d'un **spécificateur de type** et d'une **liste d'identificateurs** séparés par une virgule est une **déclaration**, par exemple :
 int a;
 int b = 1, c;
 char message[80];
- Toute **variable** doit être **déclarée** avant d'être **utilisée** en C.

Structure d'un programme C

- Un programme C se présente de la façon suivante :
 [directives au préprocesseur]
 [déclarations de variables externes]
 [fonctions secondaires]
 main ()
 {
 déclarations de variables internes
 instructions
 }

Préprocesseur

- Le **préprocesseur** effectue un **prétraitement** du programme source avant qu'il soit **compilé**.
- Ce préprocesseur **exécute** des **instructions particulières** appelées **directives**.
- Ces directives sont **identifiées** par le caractère **#** en tête.
- **Inclusion de fichiers**
#include <nom-de-fichier> /* répertoire standard */
#include "nom-de-fichier" /* répertoire courant */

- La gestion des fichiers (**stdio.h**) /* Entrees-sorties standard */
- Les fonctions mathématiques (**math.h**)
- Traitement de chaînes de caractères (**string.h**)
- Le traitement de caractères (**ctype.h**)
- Utilitaires généraux (**stdlib.h**)
- Date et heure (**time.h**)

Directives du préprocesseur

- `#define chaine1 chaine2`

remplacement littéral de la chaîne de caractères
chaine1 par **chaine2**

```
/*  
 * RAND_MAX is the maximum value that may be returned by rand.  
 * The minimum is zero.  
 */  
#define RAND_MAX      0x7FFF  
  
/*  
 * These values may be used as exit status codes.  
 */  
#define EXIT_SUCCESS   0  
#define EXIT_FAILURE   1  
  
#define __need_size_t  
#define __need_wchar_t
```


Directives du préprocesseur

```
#ifndef _STDLIB_H_  
#define _STDLIB_H_  
....  
#endif  
#if !defined (__STRICT_ANSI__)  
#else
```

Quelques lignes du fichier stdlib.h

1^{er} Programme

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main()
4  {
5      printf("Bonjour les GIs !!!!");
6      return EXIT_SUCCESS;
7  }
8
9
```

Expliquer, ligne par ligne, la signification de tous les termes de ce programme

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main()
4  {
5      printf("Bonjour les GIs !!!!");
6      return EXIT_SUCCESS;
7      system("pause");
8  }
9
```

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main()
4  {
5      printf("Bonjour les GIs !!!!");
6      system("pause");
7      return EXIT_SUCCESS;
8  }
9
```

Expliquer la différence entre ces deux codes

La fonction `printf()`

Librairie : **stdio.h**

#include <stdio.h>

Syntaxe : **int** **printf**(const char *format [, arg [, arg]...]);

Description : Permet l'écriture formatée (l'écran par défaut).

Exemple :

```
printf("Qu'il est agreable d'utiliser printf en\t C,\nlorsqu'on l'utilise  
\"proprement\".\n");
```

Résultat sur la sortie :

*Qu'il est agreable d'utiliser printf en C,
lorsqu'on l'utilise "proprement".*

Les caractères précédés de \ sont interprétés comme suit :

- `\\` : caractère \
- `\n` : retour à la ligne
- `\t` : tabulateur.
- `\"` : caractère "

`\a` : sonnerie

La fonction printf()

Les constantes de type caractère ont une valeur entière dans la table ASCII

```
char c1 = 'A',
```

```
c2 = '\x41'; /* représentation hexadécimale */
```

caractères	nom	symbol	code hexa	décimal
\n	newline	A	10	
\t	tabulation		9	9
\b	backspace		8	8
\\	backslash	\	5C	92
\'	single quote	'	27	39
\"	double quote	"	22	34

La fonction scanf()

Librairie : **stdio.h**.

#include <stdio.h>

Syntaxe : `int scanf(const char *format [, arg [, arg]...]);`

Description : Lit à partir de **stdin** (clavier en principe), les différents arguments en appliquant le **format spécifié**.

Exemple : `scanf(" %d", &age);` /* lecture de l'âge, on donne l'adresse de age */

Ces fonctions utilisent des **formats** qui permettent de lire/écrire des variables de différents types : **Format des paramètres passés en lecture et écriture.**

"%c" : lecture d'un caractère.

"%d" ou **"%i"** : entier signé.

"%e" : réel avec un exposant.

"%f" : réel sans exposant.

"%g" : réel avec ou sans exposant suivant les besoins.

"%G" : identique à g sauf un E à la place de e.

"%o" : le nombre est écrit en base 8.

"%s" : chaîne de caractère.

"%u" : entier non signé.

"%x" ou **"%X"** : entier base 16 avec respect majuscule/minuscule.

scanf ()

- Sert à **la lecture de données** et **convertit** la succession de caractères donnés en entiers, flottants, caractères, chaîne de caractères
- Syntaxe :
 - scanf (format,arg1,arg2,.....,argn)
 - le nombre d'arguments est **quelconque**
 - arg1, arg2, ..., argn sont les **adresses** des variables dans lesquelles on stocke les valeurs lues
 - variable simple (entier,caractère,flottant) : **&v**
 - chaîne de caractères = tableau : v
 - le format est une **chaîne de caractères** précisant le type des arguments afin de **convertir** la suite de caractères lus dans les arguments

Scanf : format

- Format

chaîne de caractères composée de caractères % suivis d'une lettre et éventuellement séparés par des blancs
la lettre indique le type de conversion à effectuer

exemple :

```
int i; float x;  
scanf("%d %f", &i, &x);
```



le %d indique que le premier argument est un entier

le %f indique que le second est un réel

réponse : 23 12.6

23 est converti en entier et stocké dans i

12.6 est converti en flottant et stocké dans x

Scanf : format

- Exemple

```
char t[20];  
int i ; float x;  
scanf ("%s %d %f", t,&i,&x);
```

réponses :

1/ abcde 123 0.05↵

2/ abcde 123 ↵

0.05↵

3/ abcde↵

123 ↵

0.05↵

Scanf : rôle des caractères , ↵ , tabulation, dans les réponses

- Dans les réponses
 , ↵ , tabulation servent de délimiteurs pour les valeurs **numériques et les chaînes de caractères (pas pour les caractères)**

- Exemples

```
scanf ("%d%f",&i,&x);
```

```
rep1 : 123          456 ↵          i = 123 , x = 456.0
```

```
rep2 : 123456 ↵          i = 123456 , x : pas encore lu (en attente)
```

```
scanf ("%s%d",ch,&i);
```

```
rep : abc 12          ch = "abc" , i=12
```

```
scanf ("%c%c",&c1,&c2);
```

```
rep1 : ab↵          c1= 'a' , c2 = 'b'
```

```
rep2 : a  b↵          c1= 'a' , c2 =
```

```
scanf ("%c%c%c",&c1,&c2,&c3);
```

```
rep1 : ab↵          c1= 'a' , c2 = 'b', c3= ↵
```

```
rep2 : ab↵          c1= 'a' , c2 = 'b'
```

```
c↵          ↗ c3 = ↵
```

Scanf : rôle des caractères et tabulation, dans la chaîne de format

- Lecture de valeurs numériques : aucun rôle
`scanf ("%d%f",&i,&x) ⇔ scanf ("%d %f",&i,&x)`
- Lecture de caractères : indique de sauter les `\n`, `\t` et `\f`
- Exemples

```
scanf ("%c%c%c",&c1,&c2,&c3);
```

```
rep1 : abc          c1= 'a' , c2 = 'b', c3= 'c'
```

```
rep2 : a  b  c      c1= 'a' , c2 = ' ', c3= 'b'
```

```
scanf ("%c %c %c",&c1,&c2,&c3);
```

```
rep1 : abc          c1= 'a' , c2 = 'b', c3= 'c'
```

```
rep2 : a  b  c      c1= 'a' , c2 = 'b', c3= 'c'
```

```
rep2 : a  b ↵
```

```
      c          c1= 'a' , c2 = 'b', c3= 'c'
```

Scanf : compléments

- Nombre de caractères lus
faire précéder le caractère de format du nombre de caractères
(max) désiré

Exemples

```
int i,j,k;
```

```
scanf("%3d %3d %3d",&i,&j,&k);
```

```
rep1 : 1  2  3          i=1 j=2 k=3
```

```
rep2 : 123  456  789    i=123 j=456 k=789
```

```
rep3 : 123456789        i=123 j=456 k=789
```

```
rep4 :1234  5678  9      i=123 j=4   k=567
```

```
int i; float x;char c;
```

```
scanf("%3d  %5f  %c",&i,&x,&c);
```

```
rep : 10  234.567  t          i=10 x=234.5 c='6'
```

Scanf : compléments

- Lecture d'une chaîne de caractères

```
char ch[50];  
scanf("%s",ch)  
rep : abcdefghi
```



// pas de &

'a'	'b'	'c'	'd'	'e'	'f'	'g'	'h'	'i'	0	?	?
-----	-----	-----	-----	-----	-----	-----	-----	-----	---	---	---

Le caractère 0 de fin de chaîne est ajouté automatiquement

Exercice : faire l'équivalent de `scanf("%s",ch)` à l'aide de `getchar()`

- Saut conditionnel de caractères : `%*d`, `%*f`
permet de sauter des données correspondantes dans la réponse

exemple :

```
int i,j; char c;  
scanf("%d    %*d    %c",&i,&c);  
rep1 : 12   34x      i=12   c='x'  
rep2 : 12   x        i=12   c='x'
```

scanf : compléments

- **Filtre sur chaînes de caractères**
[caractères admissibles] ou [^caractères non admissibles]

exemples


```
char ch[100];  
scanf("%[0123456789]",ch);  
rep : 32a48                ch="32"
```

```
scanf("%[^0123456789]",ch);  
rep : 32a48                ch="a"
```

raccourcis : [0123456789] ou [0-9]
 [abcdefg] ou [a-g]

Affichages et saisies

Librairie : *stdio.h*

Fonction	Syntaxe	Description
printf	printf(const char *format [, arg [, arg]...]);	Écriture formatée  sortie standard
scanf	scanf(const char *format [, arg [, arg]...]);	Lecture formatée  entrée standard
putchar	putchar(int c);	Écrire le caractère c 
getchar getch	getchar(); getch(); <conio.h>	Lecture d'un caractère 
puts gets	*puts(char *s); *gets(char *s);	Ecriture/Lecture d'une chaîne de caractères, terminée par \n
sprintf	sprintf(char *s, char *format, arg ...);	Ecrit dans la chaîne d'adresse s.
sscanf	sscanf(char *s, char *format, pointer ...);	Lit la chaîne d'adresse s.

Examples :

```
int main()
{
    int i;
    for(i=0;i<=255;i++)
        printf(" \n %d => %c : %X : %x : %o",i,(char)i,i,i,i);
}
```

```
int main()
{
    char c1; float c2, c3;
    scanf("%c%f%f",&c1,&c2,&c3);
    printf(" \n %c  %f  %f ",c1,c2,c3);
}
```

```
int main()
{
    char c;
    int i;
    float x;
    scanf("%2d %5f %c",&i,&x,&c);
    printf(" \n %d\n  %f\n  %c ",i,x,c);
}
```

22 123.3333 S

22
123.300003
3

Opérateurs d'adressage

- Adresse de : &
Syntaxe : &variable , donne l'adresse mémoire de la variable
Exemple :
`int i,adr;`
`adr = &i;`
ne pas confondre avec le "et" bit à bit
- Dont l'adresse est : *
Syntaxe *expression : donne le mot mémoire dont l'adresse est donnée par l'expression
Exemple :
`int i, ^*adr;`
`i=1;`
`adr = &i;`
`printf("%d", *adr); -> 1`

Opérateur de taille : sizeof

- Donne la taille de l'implantation
- 2 syntaxes

1/ **sizeof** expression

exemple :

```
int i,j ;
```

```
j= sizeof i ; -> 2 ou 4 (octets)
```

2/ **sizeof**(type)

exmples :

```
typedef char tab[100];
```

```
tab t;
```

```
int n;
```

```
n = sizeof(int), -> 2 ou 4 (octets)
```

```
n = sizeof(tab) -> 100 (char)
```

Opérateurs et expressions

- Opérateurs à un paramètre:
 - - change le signe de la variable
 - ~ complément à 1
 - * « *indirection* » (*pointeurs*)
 - *value = *salary; /* contenu de la variable pointé par salaire */*
 - & adresse
 - *int old, *val = &old;*
 - ++/-- incrémentation/décrémentation
 - sizeof()
 - *printf(" \n char %d ",sizeof(char));*
 - *printf(" \n long int %d ",sizeof(long int));*
 - *printf(" \n int %d ",sizeof(int));*
 - *printf(" \n short int %d ",sizeof(short int));*
 - *printf(" \n short %d ",sizeof(short));*

```
int i; scanf("%d",&i); printf("\n %d %X %d %d \n",i,i,~i,&i);  
scanf("%d",&i); printf("\n %d %x %d %d \n",i,i,~i,&i);
```

// tester avec des nombres positifs et négatifs exemple 2 -2 la representation des chiffres négatif sur un ordinateur

Opérateurs et expressions

- Opérateurs arithmétique:
 - $*, /, +, -$
 - $\%$ modulo
- Opérateurs sur bits:
 - $<<, >>$ décalage à gauche ou à droite
 - `status = byte << 4;`
 - $\&$ et
 - $|$ ou
 - \wedge ou exclusif

Remarque :

$7/2 \longrightarrow 3$

$\left. \begin{array}{l} 7.0/2 \\ 7/2.0 \\ 7.0/2.0 \end{array} \right\} \longrightarrow 3.5$

Opérateurs et expressions

Les opérateurs de comparaison

<	plus petit
<=	plus petit ou égal
>	plus grand
>=	plus grand ou égal
==	égal
!=	différent

Le type **booléen** **n'existe pas**. Le résultat d'une expression logique vaut **1** si elle est **vraie** et **0** sinon.

Les opérateurs logiques

&&	et
	ou
!	non

Réciproquement, **toute valeur non nulle** est considérée comme **vraie** et la valeur **nulle** comme **fausse**.

Exemple

```
int i;  
float f;  
char c;
```

```
i = 7;    f = 5.5;    c = 'w';
```

```
f > 5      =====> vrai (1)
```

```
(i + f) <= 1 =====> faux (0)
```

```
c == 'w'   =====> vrai (1)
```

```
c != 'w'   =====> faux (0)
```

```
(i >= 6) && (c == 'w') =====> vrai (1)
```

```
(i >= 6) || (c == 119) =====> vrai (1)
```

!expr1 : est vrai si expr1 est faux et faux si expr1 est vrai ;

expr1 && expr2 est vrai si les deux expressions expr1 et expr2 sont vraies et faux sinon. L'expression **expr2** n'est évaluée que dans le cas où l'expression **expr1** est vraie ;

expr1 || expr2 est vrai si expr1 est vrai ou expr2 est vrai. et faux sinon.

L'expression **expr2** n'est évaluée que dans le cas où l'expression **expr1** est fausse.

Opérateurs et expressions

Contractions d'opérateurs

- Il y a une famille d'opérateurs

$+=$ $-=$ $*=$ $/=$ $\%=$
 $\&=$ $|=$ $\wedge=$
 $<<=$ $>>=$

- Pour chacun d'entre eux

expression1 *op*= expression2

est équivalent à:

(expression1) = (expression1) *op* (expression2)

a += 32;

a = a + 32;

f /= 9.2;

f = f / 9.2;

i *= j + 5;

i = i * (j + 5);

Opérateurs bit à bit

- Opèrent sur les représentations des valeurs
- $\&$ et , $|$ ou, \wedge ou-exclusif, \sim complément à 1 ,
- \ll décalage à gauche, \gg décalage à droite,
- Attention : $\& \neq \&\&$

- Exemples

5 0000 0000 0000 0101

20 0000 0000 0001 0100

5 $\&$ 20 0000 0000 0000 0100 \Rightarrow 5 $\&$ 20 \Rightarrow 4

5 $|$ 20 0000 0000 0001 0101 \Rightarrow 5 $|$ 20 \Rightarrow 21

5 \wedge 20 0000 0000 0001 0001 \Rightarrow 5 \wedge 20 \Rightarrow 17

\sim 5 1111 1111 1111 1010 \Rightarrow -6

- Affectation/bit-à-bit : $\&=$, $|=$, $\wedge=$, $\sim=$

Décalages

- Décalages

- à gauche $a \ll b$: a est décalé à gauche de b bits (les bits ajoutés valent 0)

5 \ll 2 0000 0000 0001 0100 20

un décalage d'une position à gauche correspond à une multiplication par 2

- à droite $a \gg b$: a est décalé à droite de b bits (les bits insérés valent le bit de poids fort)

14 0000 0000 0000 1110

14 \gg 2 0000 0000 0000 0011 3

-6 1111 1111 1111 1010

-6 \gg 1 1111 1111 1111 1101 -3

un décalage d'une position à droite correspond à une division par 2 (en respectant le signe)

Opérateurs et expressions

Incrément et décrement

- C a deux opérateurs spéciaux pour incrémenter (ajouter 1) et décrémenter (retirer 1) des variables entières

`++` increment : `i++` ou `++i` est équivalent à `i += 1` ou `i = i + 1`

`--` decrement

- Ces opérateurs peuvent être préfixés (avant la variable) ou postfixés (après)

“i” vaudra 6

“j” vaudra 3

“i” vaudra 7

```
int i = 5, j = 4;
```

```
i++;
```

```
--j;
```

```
++i;
```

Opérateurs et expressions

Préfixe et Postfixe

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

```
int main(void)
```

```
{
```

```
    int    i, j = 3;
```

```
    i = ++j;
```

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

```
    j = 3;
```

```
    i = j++;
```

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

```
    return 0;
```

```
}
```

équivalent à:

1. j++;
2. i = j;

équivalent à:

1. i = j;
2. j++;

i=4, j=4

i=3, j=4

Opérateurs divers

- () : force l'ordre des calculs

ex : $1 + 2 * 3 \rightarrow 7$

$(1+2) * 3 \rightarrow 9$

- [] pour les tableaux

$t[2]$ équivalent à $*(t+2)$

- \rightarrow et $.$ (opérateurs sur structures, + tard)

Priorité des opérateurs

Priorité	Opérateurs	Description	Associativité
15	() [] -> .	opérateurs d'adressage	->
14	++ --	incrément/décrément	<-
	~	complément à un (bit à bit)	
	!	non unaire	
	& *	adresse et valeur (pointeurs)	
	(type)	conversion de type (cast)	
	+ -	plus/moins unaire (signe)	
13	* / %	opérations arithmétiques	->
12	+ -	""	->
11	<< >>	décalage bit à bit	->
10	< <= > >=	opérateur relationnels	->
9	== !=	""	->
8	&	et bit à bit	->
7	^	ou exclusif bit à bit	->
6		ou bit à bit	->
5	&&	et logique	->
4		ou logique	->
3	?:	conditionnel	<-
2	= += -= *= /= %= >>= <<= &= ^= =	assignations	<-
1	,	séparateur	->

Priorité des opérateurs

$a - b / c * d$

$(a-b) / (c-d)$

$i = j = k = 0;$

$!0 == ++0$

$1 == 1$

`a=-1; (!a==++a)? printf(" !a==++a"): printf("Noonn !a==++a");`

`printf("\n %d %d ", 2/2*31/2, 60*2/31*2);`

`printf("\n %d %d ", 2/2/31/2, 90/2/31*2);`

`printf("\n %d %d ", 20/2%31/2, 90%2/31*2);`

`printf("\n %d %d ", 20<<2%31/2, 100%30>>2);`

`printf("\n %d %d ", 2/2|31/2, 60*2&31*2);`

`printf("\n %d %d ", 2/2+31/2, 90/2-31*2);`

`printf("\n %d %d ", 20/2&&31/2, 90%2||31*2);`

`printf("\n %d %d ", 20&&31||!2, 100^30);`

Priorité des opérateurs (exercices)

```
main(){
int x, y , z;
x = 2;
x += 3 + 2; printf("%d\n",x);
x -= y = z = 4; printf("%d%d%d\n",x,y,z);
x = y == z; printf("%d%d%d\n",x,y,z);
x == (y = z); printf("%d%d%d\n",x,y,z);

x = 3; y = 2 ; z = 1;
x = x && y || z ; printf("%d\n", x);
printf ("%d\n", x || !y && z);
x = y = 0;
z = x ++ -1; printf ("%d, %d\n", x, z);
z += -x ++ + ++ y; printf ("%d, %d\n", x, z);

x = 1 ; y = 1;
printf("%d\n", ! x | x);
printf("%d\n", ~ x | x);
printf("%d\n", x ^ x);
x <=<= 3 ; printf("%d\n", x);
y <=<= 3 ; printf("%d\n", y);
y >=>= 3 ; printf("%d\n", y);
```

Priorité des opérateurs (exercices)

```
x = 0 ; y = 0; z = 0;
x += y += z;
printf("%d\n", x < y ? y : x) ;
printf("%d\n", x < y ? x++ : y++ ) ;
printf("%d, %d\n", x , y);
printf("%d\n", z += x < y ? x++ : y++ ) ;
printf("%d, %d\n", y , z);
```

```
x = 3; y = z = 4;
printf("%d\n", ( z >= y >= x ) ? 1 : 0) ;
printf("%d\n", z >= y && y >= x ) ;
x = y = z = 0;
}
```

Opérateurs et expressions

- Opérateur conditionnel:

L'instruction suivante :

```
result = (mode > o) ? 1 : o;
```

Est équivalente à :

```
if (mode > o)
    result = 1;
else
    result = o;
```


Les structures de contrôle en C

Alternative:

if-else

Choix Multiple:

switch-case

Itérations:

for, while, do-while

Rupture de Contrôle:

break, continue, return ...
goto

Les tests

- Syntaxes :

```
if (expression_test)
    bloc_instructions_1
```

Si `expression_test` est vraie on exécute le bloc d'instructions 1, sinon on passe à la suite.

```
if (expression_test)
    bloc_instructions_1
else
    bloc_instructions_2
```

Si `expression_test` est vraie on exécute le bloc d'instructions 1 sinon on exécute

Tests (suite)

- Enchaînement de *if*:

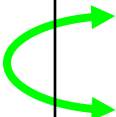
```
if (expression_test1)
    bloc_d_instructions_1
else if (expression_test2)
    bloc_d_instructions_2
    else if (expression_test3)
        bloc_d_instructions_3
    ...
    else
        bloc_d_instructions_final
```

Tests (suite)

- **else** est associé avec le **if** le plus proche

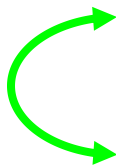
```
int i = 100;

if(i > 0)
    if(i > 1000)
        printf("i > 1000\n");
    else
        printf("i is reasonable\n");
```



```
int i = 100;

if(i > 0) {
    if(i > 1000)
        printf(" i > 1000 \n");
} else
    printf("i is negative\n");
```



Expressions évaluées dans les tests

- `if (a = b)` : **erreur fréquente**, expression toujours vraie !
- `if (a == b)` : a égal à b
- `if (a != b)` : a différent de b
- `if (a > b)` : a supérieur à b
- `if ((a >= b) && (a > 0))` : a **supérieur ou égal** à b **et** a positif
- `if ((a <= b) || (a > 0))` : a **inférieur ou égal** à b **ou** a positif
- Tout ce qui est `0` (`'0'` `0.0000` `NULL`) est faux
- Tout ce qui est `!= 0` (`1` `'0'` `0.0001` `1.34`) est vrai
- `if(32) printf("ceci sera toujours affiche\n");`
- `if(0) printf("ceci ne sera jamais affiche\n");`

`if(delta != 0) ⇔ if(delta)`

`if(delta == 0) ⇔ if(!delta)`

```
if (a<b) {  
    min=a;  
}  
else {  
    min=b;  
}
```

Exemples :

```
if (i < 10) i++;
```

La variable `i` ne sera incrémentée que si elle a une valeur inférieure à 10.

```
if (i == 10) i++;
```

== et pas =

La variable `i` ne sera incrémentée que si elle est égale à 10.

```
if (!recu) printf ("rien reçu\n");
```

Le message "rien reçu" est affiché si reçu vaut zéro.

```
if ((!recu) && (i < 10)) i++;
```

i ne sera incrémentée que si recu vaut zéro et $i < 10$.

Si plusieurs instructions, il faut les mettre entre accolades.

```
if ((!recu) && (i < 10) && (n!=0) ){
    i++;  moy = som/n;
    printf(" la valeur de i =%d  et moy=%f\n", i,moy) ;
}
else {
    printf ("erreur \n"); i = i +2;           // i +=2 ;
}
```

Boucle « for »

- La boucle **for** :

```
for (initialisation ; test ; instruction) {  
    instructions;  
}
```

- Exemple :

```
for (i = 0 ; i <= 50 ; i++) {  
    printf(" i = %d\n ",i);  
}
```

« Commencer à $i = 0$, tant que $i \leq 50$, exécuter l'instruction `printf` et incrémenter i »

Boucle « for »

```
/* Boucle for */
#include <stdio.h>
#define NUMBER 22
main()
{
    int count, total = 0;

    for(count = 1; count <= NUMBER; count++, total += count)
        printf("Vive le langage C !!!\n");
    printf("Le total est %d\n", total);
}
```

Initialisation

Condition de fin
de boucle

Incrémentation et autres fonctions

Exemples

```
double angle;
```

```
for(angle = 0.0; angle < 3.14159; angle += 0.2)  
    printf("sine of %.1lf is %.2lf\n",angle, sin(angle));
```

```
int i, j, k;
```

```
for(i = 0, j = 2, k = -1; (i < 20) &&(j==2); i++, k--)
```

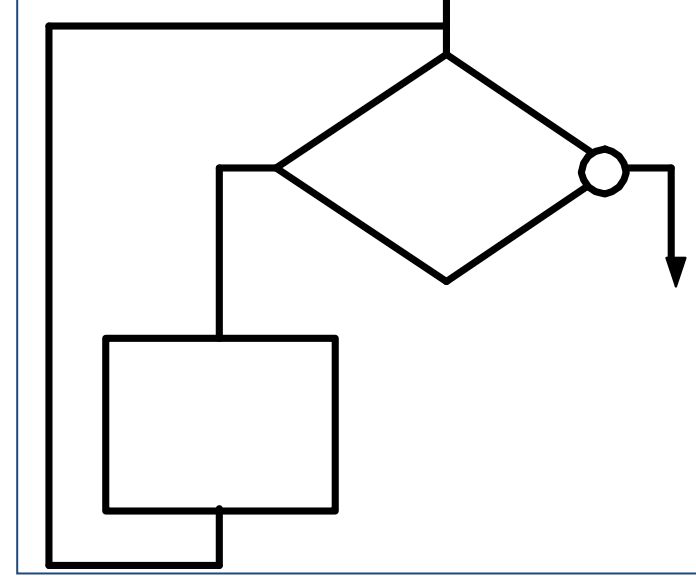
```
for( ; ; )  
{  
    .....; /* bloc d'instructions */  
    .....;  
    .....;  
}
```

est une boucle infinie (répétition infinie du bloc d'instructions).

Boucle while

- La boucle **while** :

```
while(test) {  
    instructions;  
}
```



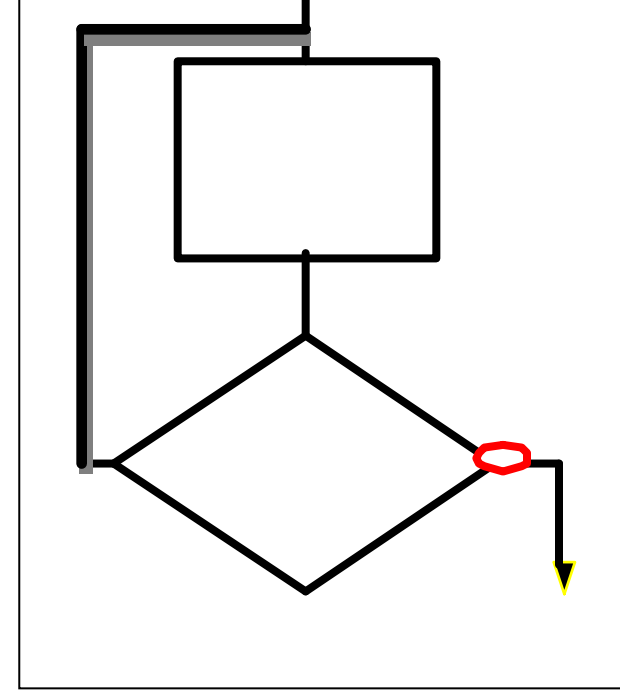
- Le test se fait d'abord, le bloc d'instructions n'est pas forcément exécuté.
- Rq: les {} ne sont pas nécessaires lorsque le bloc ne comporte qu'une seule instruction.
- exemple

```
...  
int i;  
i = 0;  
while (i < 10)  
{  
    printf("i = %d \n ",i);  
    i++;  
}
```

« Tant que i est inférieur à 10, écrire i à l'écran, incrémenter i »

Boucle do ... while

- La boucle do ... while :
do {
 instructions;
} while (test);
- permet d'exécuter au moins une fois les instructions avant d'évaluer le test



Boucle « while »

```
/* Boucle while */  
#include <stdio.h>  
#define NUMBER 22  
main()  
{
```

Initialisation

```
    int count = 1, total = 0;
```

Condition de fin de boucle
(boucle tant que vrai)
(boucle faite que si vrai)

```
    while(count <= NUMBER)
```

```
    {  
        printf("Vive le langage C !!!\n");  
        count++;  
        total += count;
```

Incrémentation

```
    }  
    printf("Le total est %d\n", total);
```

```
}
```

Boucle do ... while

```
/* Boucle do while */  
#include <stdio.h>  
#define NUMBER 22  
main()  
{
```

Initialisation

```
    int count = 1, total = 0;
```

```
do
```

Incrémentation

```
{  
    printf("Vive le langage C !!!\n");  
    count++;  
    total += count;  
} while(count <= NUMBER);  
printf("Le total est %d\n", total);
```

Condition de fin de boucle
(boucle tant que vrai)
(boucle faite au moins 1 fois)

```
}
```

Choix multiple: switch case

```
/* Utilisation de switch case */
```

```
main()
```

```
{
```

```
  char choix;
```

```
  ...
```

```
  switch(choix)
```

```
{
```

```
  case 'a' : fonctionA();
```

```
  case 'b' : fonctionB();
```

```
  case 'c' : fonctionC();
```

```
  default : erreur(3);
```

```
}
```

```
}
```

Paramètre de décision

Exécuté si choix = a

Exécuté si choix = a ou b

Exécuté si choix = a, b ou c

**Exécuté si choix non
répertorié par un « case »
et si choix = a, b ou c**

Effet du « break »

/* Utilisation de switch case */

main()

{

char choix;

...

switch(choix)

{

case 'a' : fonctionA(); break;

case 'b' : fonctionB(); break;

case 'c' : fonctionC(); break;

default : erreur(3);

}

}

Paramètre de décision

Exécuté si choix = a

Exécuté si choix = b

Exécuté si choix = c

**Exécuté si choix non
répertorié par un « case »**

switch = AU CAS OU ... FAIRE ...

```
switch(variable de type char ou int)    /* au cas où la variable vaut: */
{
  case valeur1: .....;                /* cette valeur1(étiquette): exécuter ce bloc d'instructions.*/
    .....;
    break;                             /* L'instruction d'échappement break;
                                       permet de quitter la boucle ou l'aiguillage le plus proche.
                                       */

  case valeur2:.....;                 /* cette valeur2: exécuter ce bloc d'instructions.*/
    .....;
    break;

  .
  .
  .
  default: .....;                     /* aucune des valeurs précédentes: exécuter ce bloc
    .....;                           d'instructions, pas de "break" ici.*/
}
```

Le bloc "default" **n'est pas obligatoire**. valeur1, valeur2, doivent être des expressions constantes. L'instruction switch correspond à une cascade d'instructions if ...else

Cette instruction est commode pour les "menus"

Instructions d'échappement

Pour rompre le déroulement séquentiel d'une suite d'instructions

Break;



```
int i, j=1;
char a;
for (i = -10; i <= 10; i++){
    while(j!=0) /* boucle infinie */
    {
        a=getchar();
        if(a== 'x')
            break;
    }
}
```

Si x est tapée au clavier

Continue;



```
for (i = -10; i <= 10; i++)
{
    if (i == 0)
        continue;
    // pour éviter la division par zéro
    printf(" %f", 1 / i);
}
```

return (expression);
permet de sortir de la fonction qui la contient

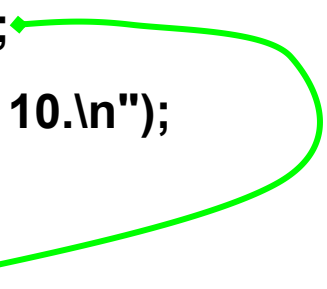
exit (expression); La fonction est interrompu.
expression : un entier indiquant le code de terminaison
du **processus**

printf, scanf : exemples

```
1  int main()
2  {
3      const int nbr_alg=2;
4      int choix,i;
5      short dd=1,deca= 1;
6      do
7      {
8          system("cls");
9          printf("\nCe programme regroupe les diff exemples de la partie algorithmique : \n\
10         les algos present jusqu'a maintenant sont %d: \
11         \n\t choix 1 : Algorithme EleveAuCarre \
12         \n\t choix 2 : Algorithme tableau de simulation \
13         \n\t choix 0 : Arrêter le programme \
14         ",nbr_alg);
15         do
16         {
17             printf("\n\t\t donner votre choix :      ");
18             scanf("%d",&choix);
19         } while(choix<0 || choix>nbr_alg);
20         switch(choix)
21         {
22             case 1 :
23                 EleveAuCarre();
24                 break;
25             case 2 :
26                 exemple_tableau_de_simulation();
27                 break;
28             case 0 :
29                 system("cls");
30                 printf("\n\n\n a bientot \n\n\n");
31                 return 0;
32             }
33         } while(1);
34     return 0;
35 }
```

goto étiquette

```
1  #include <stdio.h>
2  void main()
3  {
4      int i, j;
5      for (i=0; i < 10; i++)
6          for (j=0; j < 4; j++) {
7              if ( (i*j) == 10)
8                  goto trouve;
9              printf("i*j != 10.\n");
10         }
11     trouve:
12     printf("i*j = %d * %d = %d == 10.\n", i, j, i*j);
13 }
```



```

deca<<0 : 1
deca<<1 : 2
deca<<2 : 4
deca<<3 : 8
deca<<4 : 16
deca<<5 : 32
deca<<6 : 64
deca<<7 : 128
deca<<8 : 256
deca<<9 : 512
deca<<10 : 1024
deca<<11 : 2048
deca<<12 : 4096
deca<<13 : 8192
deca<<14 : 16384
deca<<15 : 32768
deca<<16 : 65536
deca<<17 : 131072
deca<<18 : 262144
deca<<19 : 524288
deca<<20 : 1048576
deca<<21 : 2097152
deca<<22 : 4194304
deca<<23 : 8388608
deca<<24 : 16777216
deca<<25 : 33554432
deca<<26 : 67108864
deca<<27 : 134217728
deca<<28 : 268435456
deca<<29 : 536870912
deca<<30 : 1073741824
deca<<31 : -2147483648
deca<<32 : 1
deca<<33 : 2
deca<<34 : 4
deca<<35 : 8
deca<<36 : 16
deca<<37 : 32
deca<<38 : 64
deca<<39 : 128
deca<<40 : 256
deca<<41 : 512
deca<<42 : 1024
deca<<43 : 2048
deca<<44 : 4096
deca<<45 : 8192
deca<<46 : 16384
deca<<47 : 32768
deca<<48 : 65536
deca<<49 : 131072
deca<<50 : 262144
deca<<51 : 524288
deca<<52 : 1048576
deca<<53 : 2097152
deca<<54 : 4194304

```

```

short deca=1;
for(i=0;i<64;i++)
printf("\n deca<<%d : %d ",i, deca<<i);

```

```

deca<<0 : 1
deca<<1 : 2
deca<<2 : 4
deca<<3 : 8
deca<<4 : 16
deca<<5 : 32
deca<<6 : 64
deca<<7 : 128
deca<<8 : 256
deca<<9 : 512
deca<<10 : 1024
deca<<11 : 2048
deca<<12 : 4096
deca<<13 : 8192
deca<<14 : 16384
deca<<15 : -32768
deca<<16 : 0
deca<<17 : 0
deca<<18 : 0
deca<<19 : 0
deca<<20 : 0
deca<<21 : 0
deca<<22 : 0
deca<<23 : 0
deca<<24 : 0
deca<<25 : 0
deca<<26 : 0
deca<<27 : 0
deca<<28 : 0
deca<<29 : 0
deca<<30 : 0
deca<<31 : 0
deca<<32 : 1
deca<<33 : 2
deca<<34 : 4
deca<<35 : 8
deca<<36 : 16
deca<<37 : 32
deca<<38 : 64
deca<<39 : 128
deca<<40 : 256
deca<<41 : 512
deca<<42 : 1024
deca<<43 : 2048
deca<<44 : 4096
deca<<45 : 8192
deca<<46 : 16384
deca<<47 : -32768
deca<<48 : 0
deca<<49 : 0
deca<<50 : 0
deca<<51 : 0
deca<<52 : 0
deca<<53 : 0
deca<<54 : 0

```

```

short dd=1,deca=1;
for(i=0;i<64;i++){
deca=dd; deca=deca<<i;
printf("\n deca<<%d : %d ",i, deca);
}

```

Langage C

Les Tableaux && Les Fonctions

Prof. A.SABOUR

Objectifs du cours ...

Objectif :

- Être capable de manipuler les Tableaux et les fonctions
- Introduire la notion de pointeur

Tableaux

- Lorsque on veut mémoriser **plusieurs** données de **même type**, on peut utiliser un **tableau** c-à-d on regroupe sous un même nom plusieurs informations
- Exemple de déclaration d'un tableau d'entiers

```
int tab[100];
```

int : type des éléments du tableau

tab : identificateur (nom du tableau)

100 : nombre d'éléments du tableau (dimension)

tab peut recevoir **100** entiers identifiés de **0** à **99** (**attention** !)

le premier est tab[0]

le second tab[1]

..

le dernier tab [99]

Tableaux

- Utilisation

chaque élément du tableau est **accessible** par un **indice** qui doit être de type **entier**, **quelque** soit le type des **éléments** du **tableau**

exemples :

int i ;

tab[2] 3^{eme} élément du tableau

tab[2+3] 6^{eme} élément du tableau

tab[i] i+1^{eme} élément du tableau

- Exemples :

stocker les 100 premiers nombres pairs : 0,2,4,...,196,198

```
int i, t[100];
```

```
for (i=0; i < 100; i=i+1)
```

```
    t[i]= 2*i;
```


Tableaux

- Remarques:

- 1/ chaque élément du tableau **s'utilise** comme une **variable**

- `tab[3] = 2;`

- 2/ le nombre maximum d'éléments du tableau (**dimension**)

- 1/ doit être **fixé** à la compilation

- 2/ ne peut être **modifié** pendant l'exécution

- 3/ **Pas d'opérations** sur les **tableaux** en tant que tels

Parcours des éléments d'un tableau

Parcours du premier au dernier

```
int i; /* l'indice de balayage doit être un entier */  
float t[100]; /* le type du tableau est quelconque */  
for (i=0; i < 100; i=i+1) // ou bien for (i=0; i <= 99; i=i+1)  
    t[i]= .....
```

Parcours du dernier au premier

```
int i; /* l'indice de balayage doit être un entier */  
float t[100]; /* le type du tableau est quelconque */  
for (i=99; i >= 0; i=i-1) // ou bien for (i=0; i <= 99; i=i+1)  
    t[i]= .....
```

La dimension

Bonne pratique de programmation

```
int i;  
int t[100];  
for (i=0; i < 100; i=i+1)  
    t[i]= 100;
```

Pb : changement de la taille du tableau

```
#define TAILLE 100  
int i;  
int t[TAILLE];  
for (i=0; i < TAILLE; i=i+1)  
    t[i]= 100;
```

Il suffit de changer TAILLE

Exemples

Point de l'espace

1ere solution :

```
float x,y,z;
```

2eme solution

```
float pt[3];
```

```
// pt[0] pour x, pt[1] pour y, pt[2] pour z
```

Mémorisation des 100 premiers nombres pairs et impairs:

```
int pairs[100], impairs[100];
```

```
int i;
```

```
for (i=0; i<100;i=i+1) {
```

```
    pairs[i]=2*i;
```

```
    impairs[i]=2*i+1;
```

```
}
```

Chaînes de caractères

- En c, **pas** de type prédéfini chaîne de caractères. En pratique on utilise des tableaux de caractères.
- Convention : le dernier caractère utile est suivi du caractère `\0` (de code ascii 0)

- Exemples :

`char t[10];` (9 caractères max, puisque une case réservée pour `\0`;
`strcpy(t,"abcdefghi");`

'a'	'b'	'c'	'd'	'e'	'f'	'g'	'h'	'i'	'\0'
-----	-----	-----	-----	-----	-----	-----	-----	-----	------

chaque lettre est accessible par l'indice

`char t[12];`
`strcpy(t,"abcdefghi");`

- Initialisation `char t[12]= "abcdefghi";` ou `char t[]= "abcdefghi";`

'a'	'b'	'c'	'd'	'e'	'f'	'g'	'h'	'i'	'\0'	?	?
-----	-----	-----	-----	-----	-----	-----	-----	-----	------	---	---

- Constante chaîne de caractères

`#define ceciestuntexte "azssddqsdqsd"`

Définitions de type utilisateur

```
typedef int entier; (entier est maintenant un type)  
entier i;
```

```
typedef int tableaude20[20];  
tableaude20 t;      // équivalent int t[20];
```

Les tableaux

Rappel : tableau =regroupement de données de même type sous un même nom, accessibles par un indice (0,...,dim-1)

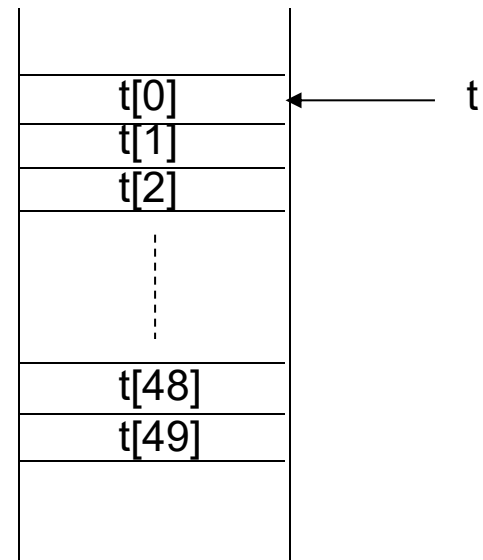
Déclaration et implantation mémoire :

`int t[50];` => réservation dans la mémoire de 50 cases contiguës d'entiers.

L'adresse de la première case est t

$\&t[0] \Leftrightarrow t$

$*t \Leftrightarrow t[0]$



Tableaux

- **Initialisation** à la compilation

int t[10] = {1,2,3,4,5,6,7,8,9,10};

float x[4] = {0.,0.25,3.14,2.57};

char couleur[4] = {'r','v','b','j'};

char texte[10] = "abcd";

int t1[10] = {1,2,3};

1	2	3	4	5	6	7	8	9	10
0.	0.25	3.14	2.57						
r	v	b	j						
a	b	c	d	\0	?	?	?	?	?
1	2	3	?	?	?	?	?	?	?

- **Dimension** par défaut:

int t[] = {0,0,0} => dimension = 3

char t[] = {'r','v','b','j'}; => dimension = 4

char t[] = "abcd" => dimension = 5

par contre **int** t[] sans **initialisation** est **incorrect**

Tableaux

- Accès aux éléments d'un tableau

```
int t[50];
```

syntaxe 1

// accès à la (i+1)ème case avec i compris entre 0 et 49
t[i];

syntaxe 2

puisque t est l'adresse de la iere case

$t[0] \Leftrightarrow *t$ // mot d'adresse t, * : opérateur mot dont
l'adresse est)

$t[1] \Leftrightarrow *(t+1)$ // rem : priorité des opérateurs)

$t[i] \Leftrightarrow *(t+i)$ //  $*t+i \Leftrightarrow t[0]+i$

Tableaux à plusieurs dimensions

- **Tableau** dont chaque case est **elle-même** un **tableau**

ex :

```
typedef int t[100] ; // t est un type  
t matrice [20];
```

matrice est un **tableau** de 20 **cases**, chacune est un tableau de 100 entiers
=> matrice est un tableau de **20*100 entiers**

autre déclaration :

```
int matrice [20][100]; // tableau de 20 "lignes" et 100 "colonnes"
```

- Accès aux éléments

par un 1er indice allant de 0 à 19 et par un 2eme indice allant de 0 à 99

matrice[3] est la 4eme case de tableau. C'est un tableau de 100 cases (entiers)

matrice[3][48] est un entier.

matrice [i][j] avec i de 0 à 19 et j de 0 à 99

matrice est un tableau à 2 dimensions

Tableaux à plusieurs dimensions

- **Pas** de limitations sur le **nombre** de dimensions

Ex à 3 dimensions : tableau de coord. de pts de l'espace

typedef float point[3] ; // x: indice 0, y : indice 1, z : indice 2

point tab[100][100]; // tab = matrice de 100 points

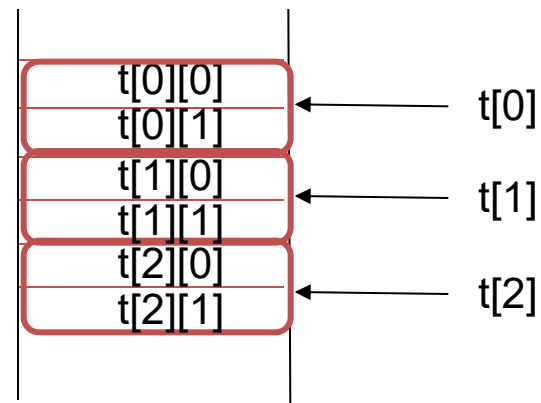
ou bien

tab[100][100][3];

tab[2][5][1] représente le "y" du point rangé en ligne 2 et colonne 5

- Implantation mémoire

int t[3][2];



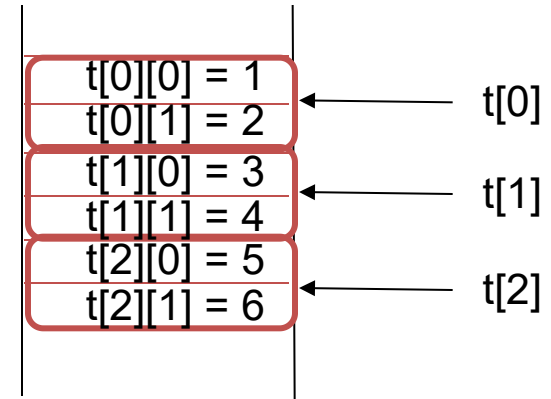
Tableaux à plusieurs dimensions

- Initialisation (à la compilation)

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

ou bien (+ clair)

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



```
int t[3][2] = {{1,2},4,{5,6}}; => t[1][1] non initialisé
```

- Initialisation grâce à des boucles

```
for (i=0;i<3;i++)
```

```
    for (j=0;j<2;j++)
```

```
        t[i][j]=0;
```

Tableaux à plusieurs dimensions

- Accès aux éléments

```
int t[dim1][dim2] ;
```

```
t[i][j] ⇔ * (*t + i*dim2+j)
```

```
int t[dim1][dim2][dim3];
```

```
t[i][j][k] ⇔ * (* (*t + i*dim2*dim3+j*dim3+k)
```

```
int t[dim1][dim2]....[dimn] ;
```

```
t[i1][i2]....[in] ⇔ * (* .... *t // n-1 *  
    +i1*dim2*dim3*dim4....*dimn  
    +i2*    dim3*dim4....*dimn  
    +.....  
    +in-1          *dimn  
    +in) )
```

=> la première dimension n'est pas utilisée dans le calcul

Tableaux à plusieurs dimensions

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

```
void main()
```

```
{
```

```
    char t[3][3] = {  
        {'1','2','3'},  
        {'4','5','6'},  
        {'7','8','9'}  
    };
```

```
    printf("value of t[0][0] : %c\n", t[0][0]);
```

```
    printf("value of t[0]   : %c\n", *t[0]);
```

```
    printf("value of t     : %c\n", **t);
```

```
}
```

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

```
int main(void)
```

```
{
```

```
    int t, i, num[3][4];
```

```
    for(t = 0; t < 3; ++t)
```

```
        for(i = 0; i < 4; ++i)
```

```
            num[ t ][ i ] = ( t * 4 ) + i + 1;
```

```
    /* now print them out */
```

```
    for(t = 0; t < 3; ++t) {
```

```
        for(i = 0; i < 4; ++i)
```

```
            printf("%3d ", num[ t ][ i ]);
```

```
        printf("\n");
```

```
    }
```

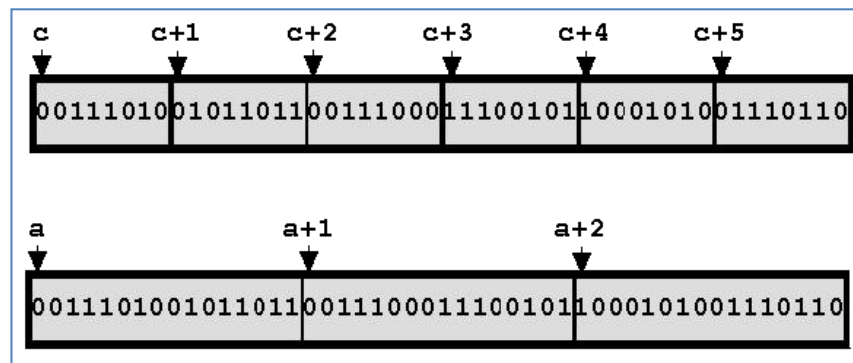
```
    return 0;
```

```
}
```

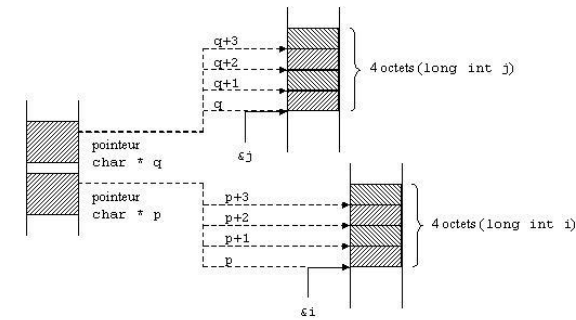
Adresse

comprendre ce qu'est une adresse.

- Lorsque l'on exécute un programme, celui-ci est **stocké en mémoire**, cela signifie que *d'une part* le **code** à exécuter est **stocké**, mais aussi que **chaque variable** que l'on a défini a **une zone** de mémoire qui lui est réservée, et la **taille** de cette **zone** correspond au type de variable que l'on a déclaré.
- En réalité la **mémoire** est constituée de plein de **petites cases de 8 bits** (un octet). Une variable, **selon** son type (donc sa taille), va ainsi occuper **une ou plusieurs** de ces **cases**. Chacune de ces « cases » (appelées **blocs**) est identifiée par un numéro. Ce numéro s'appelle **adresse**.



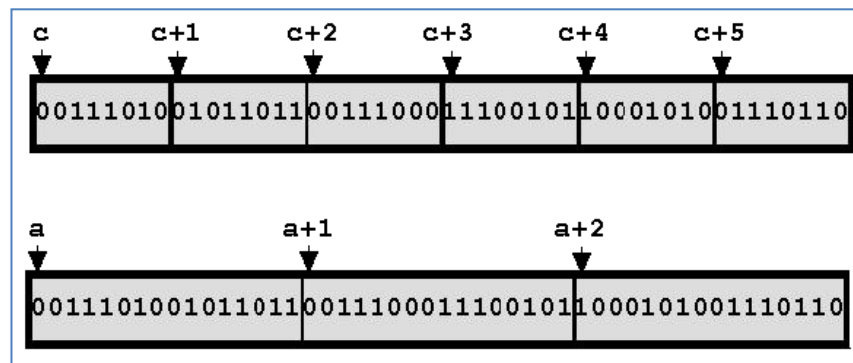
Adresse



- On utilise très souvent les adresses dans :
 - la saisie des données avec `scanf`
 - les appels des `fonctions` pour `recevoir` des `valeurs retournées`
- L'adresse (son `emplacement` en `mémoire`) d'une variable est `accessible` par l'opérateur `&` (adresse de)
 - ex : `scanf(("%d", &unEntier) ;`
 - `scanf` a `besoin` de `l'adresse` en mémoire de la variable `unEntier` pour y `placer` la `valeur lue`

Taille d'un emplacement en mémoire

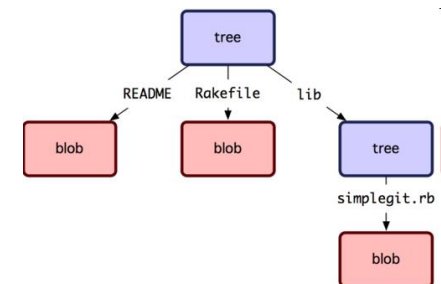
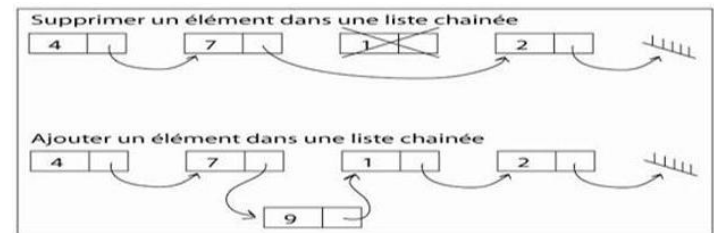
- Pour connaître la taille d'un emplacement en mémoire, nous utilisons l'opérateur **sizeof** () ;
 - Le paramètre passé est soit un **type**, soit une **variable**
 - Le résultat retourné est le **nombre d'octets** nécessaire pour stocker une valeur de ce type
- `sizeof(char) ; /* retourne 1 */`
- `sizeof(unEntier) ; /* retourne 4 si unEntier est déclaré int */`



Pointeurs

Intérêt des pointeurs

- On utilise des pointeurs dans :
 - La **transmission des arguments** par pointeurs (résultats de retour)
 - Permettre le passage par Adresse pour des paramètres des fonctions
 - La manipulation des tableaux
 - Les structures de données dynamiques (tableaux, liste linéaire chaînée, arbre binaire, ...)
 - Réaliser des structures de données récursives (listes et arbres)



Pointeurs

- Un pointeur est une **variable** contenant l'adresse d'une autre variable d'un type donné.
- La notion de pointeur est une technique de programmation **très puissante**, permettant de définir des **structures dynamiques**, c'est-à-dire qui **évoluent** au cours du temps (par opposition aux tableaux par exemple qui sont des structures de données **statiques**, dont la taille est figée à la définition).

Pointeurs

- Un **pointeur** est une variable dont la **valeur** est une **adresse**
- Un pointeur est une variable qui doit **être définie** en **précisant** le **type** de **variable pointée**, de la façon suivante :

type * Nom_du_pointeur ;

- Le type de variable pointée peut être aussi bien un **type primaire** (tel que int, char...) **qu'un type complexe** (tel que **struct**...).
- Déclaration : *type * identificateur ;*
 - ex: `int *pointeurSurEntier ;`
 - PointeurSurEntier est une variable qui peut contenir l'adresse mémoire d'une variable entière
 - `int unEntier, *pointeur = &unEntier;`

Pointeurs

- Exemple :

int *p;

- On dira que :
 - p est un pointeur sur une variable du type **int** , ou bien
 - p peut contenir l'adresse d'une variable du type **int**
 - *p est de type int, c'est l'emplacement mémoire pointé par p.
- Grâce au symbole '*' le **compilateur** sait qu'il s'agit d'une **variable** de type **pointeur** et non d'une **variable ordinaire**, de plus, étant donné que **vous précisez (obligatoirement)** le type de **variable**, le **compilateur saura combien de blocs suivent** le bloc situé à **l'adresse pointée**.

Pointeurs

- A la déclaration d'un pointeur p, il ne pointe a priori sur aucune variable précise : p est un pointeur non initialisé.
- **Toute utilisation de p devrait être précédée par une initialisation.**
- la valeur d'un pointeur est toujours un entier (codé sur **16 bits, 32 bits** ou **64 bits**).
- Pour initialiser un pointeur, le langage C fournit l'opérateur unaire **&**. Ainsi pour récupérer l'adresse d'une variable A et la mettre dans le pointeur P (P pointe vers A) : `P=&A;`

Pointeurs

*void **

*void **

- Un pointeur doit **préférentiellement** être typé !
- Il est toutefois possible de définir un pointeur sur 'void', c'est-à-dire sur quelque chose qui n'a pas de type prédéfini (`void * toto`).
- Ce genre de pointeur sert généralement de **pointeur de transition**, dans une fonction générique, avant un **transtypage** permettant d'accéder **effectivement** aux données pointées.

*void **

*void **

*void **

Pointeurs

Par exemple :

```
int a = 2;  
char b;  
int *p1;  
char *p2;  
p1 = &a;  
p2 = &b;
```

```
*p1 = 10;  
*p2 = 'a';
```

```
a = (*p1)++;
```

```
int A, B, *P;  
A = 10;  
B = 50;
```

```
P = &A ;  
B = *P ;  
*P = 20;  
P = &B;
```

- a) **int *p , x = 34; *p = x;**
- b) **int x = 17 , *p = x; *p = 17;**
- c) **double *q; int x = 17 , *p = &x; q = p;**
- d) **int x, *p; &x = p;**
- e) **char mot[10], car = 'A', *pc = &car ; mot = pc;**

Les fonctions

- On appelle fonction un **sous-programme** qui permet **d'effectuer** un **ensemble d'instructions** par simple appel de la fonction dans le corps du programme principal.
- Les fonctions permettent d'exécuter dans plusieurs parties du programme une série d'instructions, cela permet une simplicité du code et donc une taille de programme minimale. D'autre part, une fonction peut faire appel à elle-même, on parle alors de fonction récursive (il ne faut pas oublier de mettre une condition de sortie au risque sinon de ne pas pouvoir arrêter le programme...).
- Une fonction permet de :
 - Remplacer une partie qui **se répète**
 - Découper un programme en **parties isolées** -> débogage, lisibilité, etc..
- Exemples : fonctions d'E/S (scanf, printf, ...), mathématiques (sin, cos, ...)

Les fonctions

- **Organisation d'un programme :**

```
type fonction1 (arguments) {  
  Déclarations de variables et de types locaux à la fonction  
  Instructions  
}  
type fonction2 (arguments) {  
  Déclarations de variables et de types locaux à la fonction  
  Instructions  
}  
...  
void main (arguments) {  
  Déclarations de variables et de types locaux à la fonction  
  Instructions  
}
```

Exemple

Type de la valeur
de retour

Argument

```
char minus_majus (char c1) {  
    char c2; /* déclarations locales */  
    if (c1 >= 'a' && c1 <= 'z')  
        c2 = c1+'A'-'a';  
    else c2=c1;  
    return (c2);  
}
```

Instructions

```
void main() {  
    char c,majuscule;  
    printf("Donner un caractere\n");  
    c = getchar(); getchar();  
    majuscule = minus_majus(c);  
    printf ("La majuscule de %c est %c\n",c,majuscule);  
}
```

Appel de la fonction

Code ASCII

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	 	Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	\$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	((72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051))	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[[123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174	|	
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135]]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177		DEL

Source: www.asciitable.com

Définition de fonction : syntaxe

```
type_fonction nom_fonction (type_arg1 arg1, ..., type_argn argn) {  
    ...  
    return (valeur retournée);  
}
```

Dans l'exemple précédent :

type_fonction : char, c'est le type de la valeur renvoyée par return
nom_fonction : minus_majus

Le nombre d'arguments est quelconque, éventuellement aucun, les parenthèses doivent toujours figurer (ex: main ())

Type de la fonction

- Une fonction peut ne pas renvoyer de valeur.
- Exemple

```
void print_majus (char c1) {  
    char c2;  
    if (c1 >= 'a' && c1 <= 'z')  
        c2 = c1+'A'-'a';  
    else c2=c1;  
    printf("la majuscule de % est %c, c1, c2);  
    return; /* ou bien return (); */  
}
```

- Dans ce cas, le type de la fonction est : void
- Le type de la fonction peut être :
 - int, float, char, ou adresse ...

Instruction return

1/ Indique la valeur de retour de la fonction.

2/ Arrête l'exécution de la fonction

```
char minus_majus (char c1) {
```

```
    if (c1 >= 'a' && c1 <= 'z')  
        return (c1+'A'-'a');  
    else return (c1);
```

```
}
```

Pour les fonction de type void, return est optionnel

```
void print_majus (char c1) {
```

```
    char c2;  
    if (c1 >= 'a' && c1 <= 'z')  
        c2 = c1+'A'-'a';
```

```
    else c2=c1;  
    printf("la majuscule de % est %c, c1, c2);
```

```
}
```

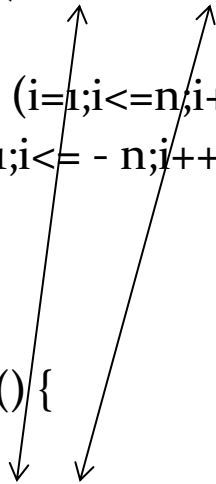
Appel des fonctions

- L'appel d'une fonction se fait en donnant son nom, suivi de la liste des **paramètres** entre parenthèses. L'ordre des paramètres correspond à celui des arguments.

- Exemple

```
float puiss (float x, int n) {  
    float y=1.0;  
    if (n>0) for (i=1;i<=n;i++) y = y*x;  
    else for (i=1;i<= - n;i++) y = y/x;  
    return (y);  
}
```

```
void main () {  
    float z,t;  
    z = puiss(10.7,2);  
    t = puiss (z, -6);  
    ...  
}
```



Appel des fonctions

- Un appel de fonction peut se faire comme opérande d'une expression, soit comme paramètre d'un autre appel de fonction.

- Exemple

```
int maximum (int x, int y) {  
    return((x>y)?x:y);  
}  
  
void main () {  
    int v1,v2,v3,m1;  
    scanf("%d%d%d", &v1,&v2,&v3);  
    m1 = maximum(v1,v2);  
    m1 = maximum(m1,v3);  
    printf("valeur maximale %d\n", m1);  
}
```

ou bien

```
m1 =maximum(v1,v2);  
printf("valeur maximale %d\n", maximum(m1,v3));
```

ou bien

```
printf("valeur maximale %d\n", maximum(maximum(v1,v2),v3));
```

Règles de déclaration et d'appel

- Toute fonction ne **peut appeler** que des **fonctions déclarées** avant **elle** ou **elle-même** (la fonction **main** ne peut pas **s'appeler**).

```
... f1 (..) {
```

```
...
```

```
}
```

```
... f2 (...) {
```

```
...
```

```
}
```

```
... f3 (...) {
```

```
...
```

```
}
```

```
void main (...) {
```

```
...
```

```
}
```

la fonction main peut appeler f1,f2,f3

la fonction f3 peut appeler f1,f2,f3

la fonction f2 peut appeler f1, f2

la fonction f1 peut appeler f1

- Lorsqu'une fonction s'appelle elle-même, on dit qu'elle est "**récursive**".

Déclarations en "avance"

- Règle précédente **contraignante**
- Solution : Prototype

En **début** de **programme** on donne le **type** de **chaque fonction** , son nom, le nombre et les types des arguments

- Information suffisante pour le compilateur.

```
float puiss (float,int);
```

```
/*Prototype ou declaration de la fonction puiss*/
```

```
void main(){
```

```
    puiss (10.2, 5);
```

```
/*Appel avant definition*/
```

```
    ...}
```

```
float puiss (float x, int n){
```

```
/*la definition de la fonction */
```

```
    float y=1.0;
```

```
    if (n>0) for (i=1;i<=n;i++) y = y*x;
```

```
    else for (i=1;i<=n;i++) y = y/x;
```

```
    return (y);
```

```
}
```

Fichier "header"

- Conseil de programmation :
Dans un fichier ".h" déclarer les **prototypes** de toutes les fonctions, par exemple malib.h
Dans le ou les fichiers ".c", insérer la directive
`#include "malib.h"`

Passage des paramètres

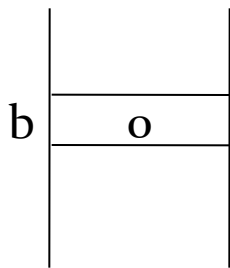
- Rappel : les **paramètres** sont associés aux **arguments** suivant l'ordre de déclaration.
- En c, cette association se fait par **COPIE** de la valeur du **paramètre** dans **l'argument**. **Chaque argument est en fait une variable locale de la fonction**. La fonction **travaille** sur **l'argument**.
- Conséquence : Une fonction ne modifie pas les paramètres d'appels

```
void f (int a){  
    a=a+1;  
}  
void main(){  
    int b;  
    b=0;  
    f(b);  
    printf("%d\n",b);  
}
```

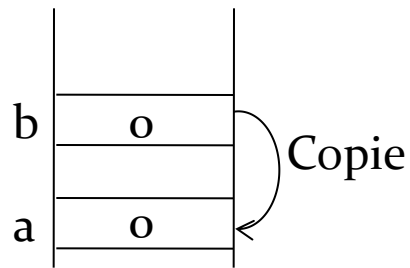
->0

Détail

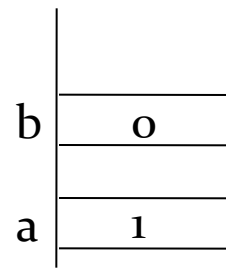
```
void f (int a){  
  a=a+1; /*3*/  
}  
void main(){  
  int b;  
  b=0; /*1*/  
  f(b); /*2*/  
  printf("%d\n",b); /*4*/  
}
```



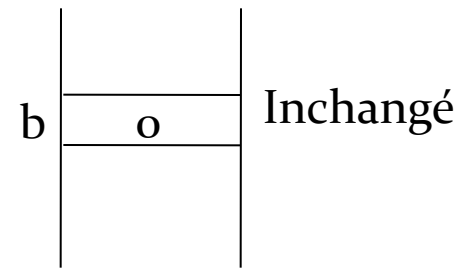
/*1*/



/*2*/



/*3*/



/*4*/

Modification des paramètres

- Si l'on veut qu'une fonction **modifie** un **paramètre**, on ne passe pas la variable **mais l'adresse** de la variable. Il y a copie de l'adresse de la variable. Dans la fonction on va chercher la **variable** par son **adresse**.

- Rappels :

opérateur **&** : & variable -> adresse de la variable

opérateur ***** : * adresse -> valeur qui se trouve à cette adresse

```
int i;
```

```
int * adresse_i; /* déclaration d'une adresse d'entier */
```

```
i=0;
```

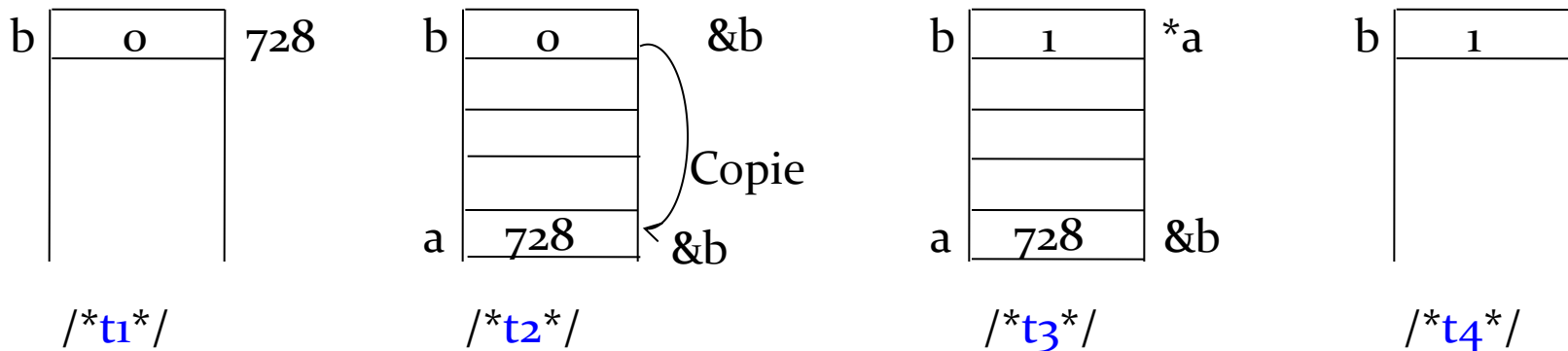
```
adresse_i=&i;
```

```
printf("%d\n",i); -> 0;
```

```
...
```

Modification des paramètres

```
void f2 (int * a){ // a est l'adresse, *a est l'entier
*a=*a+1;           /*t3 on incrémente le mot d'adresse a*/
}
void main(){
int b;
b=o;               /*t1*/
f(&b);             /*t2 &b est l'adresse de b */
printf("%d\n",b);  /*t4*/ -> 1
}
```



Exemple : `scanf ("%d",&v);`

Passage d'un tableau à une dimension en paramètre

- Rappels:
 - Lorsqu'on **déclare** un **tableau**, par ex `int t[10]`, `t` est **l'adresse** du 1^{er} élément du tableau
 - **Chaque élément** du tableau peut être **accédé** par `t[i]` ou `*(t+i)`
 - La première **dimension** d'un tableau n'est pas utilisée dans le calcul de l'adresse d'une case

- Exemple

```
void printtab (int t1[50]) {  
    int i;  
    for (i=0;i<50;i++)  
        printf("%d",t1[i]);  
}  
  
void main () {  
    int t[50];  
    .....  
    printtab(t);  
}
```

Passage d'un tableau à une dimension en paramètre

- Puisque la dimension n'est pas utilisée, on **peut ne pas** la donner

```
void printtab (int t1[50]){  
    int i;  
    for (i=0;i<50;i++)  
        printf("%d",t1[i]);  
}
```

ou bien

```
void printtab (int t1[]) {  
    int i;  
    for (i=0;i<50;i++)  
        printf("%d",t1[i]);  
}
```



Syntaxes équivalentes

Passage d'un tableau à une dimension en paramètre

Conséquence : on **peut** donc appeler cette **fonction** avec tout **tableau d'entiers** quelle que soit sa **dimension**. C'est au programmeur à gérer les **débordements** de **tableau** => donner le nombre de cases sur lequel travaille la fonction

```
void printtab (int t1[], int n){  
    int i;  
    for (i=0;i<n;i++)  
        printf("%d",t1[i]);  
}
```

```
void main () {  
    int t[50],t2[100];  
    ...  
    printtab(t,50); /*affiche toutes les cases de t de 0 à 49*/  
    printtab(t2,100); /*affiche toutes les cases de t2 de 0 à 99*/  
    printtab(t+20,30);/*affiche toutes les cases de t de 20 à 49*/  
    printtab(t+20,10);/*affiche toutes les cases de t de 20 à 30*/  
}
```

Passage d'un tableau à une dimension en paramètre

Puisqu'en fait `t1` est une **adresse**, on peut le **déclarer comme tel**

```
void printtab (int * t1,int n) {  
    int i;  
    for (i=0;i<n;i++)  
        printf("%d",t1[i]);  
}
```

Passage d'un tableau à une dimension en paramètre

Conséquence :

Si un **argument** est de type **tableau** (càd une adresse), la fonction peut modifier les **cases** du tableau

```
void misea0 (int t1[], int n){
    int i;
    for (i=0;i<n;i++)
        t1[i]=0;
}

void main () {
    int t[10]={1,2,3,4,5,6,7,8,9,10};
    printtab(t,10); -> 1 2 3 4 5 6 7 8 9 10
    misea0(t,10);
    printtab(t,10); -> 0 0 0 0 0 0 0 0 0 0
}
```

Visibilité des variables

- On appelle **visibilité** ou **portée** des **variables** les règles qui **régissent l'utilisation** des **variables**. Les mêmes règles régissent les types.
- Règle 1 : **variables globales**
Les variables déclarées **avant** la 1ere **fonction** peuvent être **utilisées** dans **toutes** les **fonctions**. Ces variables sont dites globales.

```
int i;  
  
void f1 () {  
    i = i+1;  
}  
  
void main(){  
    i=0;  
    f1();  
    printf("%d\n",i) ;    //-> 1  
}
```

Visibilité des variables

- Règle 2 : **variables locales**

Les variables déclarées dans une fonction ne peuvent être utilisées que dans cette fonction. Ces variables sont dites locales.

```
void f1 () {  
    int i;  
    i = i+1;  
}  
void main(){  
i=0;  -> ERREUR : i n'existe pas pour main  
...  
}
```

Visibilité des variables

- Règle 3 : **arguments = variables locales**

Les arguments d'une fonction sont des variables locales de la fonction.

```
void f1 (int i) {  
    i = i+1; /* i est une variable locale de la fonction */  
}  
  
void main(){  
    int j=1;  
    f1(j);  
    printf("%d\n",j)  
}
```

- Règle 4 : **Au sein d'une fonction, toutes les variables doivent avoir des noms distincts**

```
void f1 () {  
    int i;  
    char i; -> ERREUR : i existe déjà  
    i = i+1;  
}
```


Visibilité des variables

- Règle 5 : Des variables déclarées dans des fonctions différentes peuvent porter le même nom sans ambiguïté.

```
void f1 () {  
    int i; ← sous-entendu i_f1  
    ...  
}  
  
void f2 () {  
    char i; ← sous-entendu i_f2  
    ...  
}  
  
void main(){  
    int i; ← sous-entendu i_main  
    ....  
}
```

Ces 3 variables n'ont rien de commun

Visibilité des variables

- Règle 6 : **Si une variable globale et une variable locale ont le même nom, on accède à la variable locale dans la fonction où elle est déclarée**

```
int i;
void f1 () {
    int i;
    i=2; /* i de f1 */
}
void main(){
    i=0; /* i global */
    f1();
    printf ("%d\n",i); -> 0
}
```

Conseils

- **Evitez** autant que possible l'usage des **variables globales** => limitation des **effets** de **bord** indésirables

```
int i;  
void f1 () {  
    ...  
    i=i+1;  
}  
void main(){  
    i=0;  
    f1();  
    printf ("%d\n",i); -> 1  
}
```

- Dans f1, on travaille sur i global :
 - Est-ce bien ce que l'on désirait (oubli de déclaration d'une nouvelle variable locale ?)
 - Débogage difficile : il faut inspecter le code en détail pour voir où sont modifiées les variables.

Conseils

- Si l'on ne peut **éviter** les **variables globales**, respecter un code pour différencier les variables globales des variables locales.

- Par exemple :

si l'initiale de la variable est une **majuscule** -> **globale** : **Vglob**

minuscule -> **locale** : **vloc**

ou bien

le nom de chaque variable globale commence par **G_** : **G_variable**

etc...

- Pas de confusion entre locales et globales.

•

permut1 && permut2

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

```
void permut1 (int a, int b)
{
    int c;
    printf("AVANT: a=%d \t b=%d \n",a,b);
    c=a;a=b;b=c;
    printf("APRES: a=%d \t b=%d \n",a,b);
}

void permut2 (int *a, int *b)
{
    int *c;
    printf("AVANT: a=%d \t b=%d \n",*a,*b);
    *c=*a;
    *a=*b;
    *b=*c;
    printf("APRES: a=%d \t b=%d \n",*a,*b);
}
```

```
main ()
{
    int u,v;
    u=2;v=3;
    permut1(u,v);
    printf("Permut1:\t u=%d \t v=%d \n",u,v);
    /*-----*/
    permut2(&u,&v);
    printf("Permut2:\t u=%d \t v=%d \n",u,v);
}
```

permut1 && permut2

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

```
void permut1 (const int a, const int b)
{
    int c;
    printf("AVANT: a=%d \t b=%d \n",a,b);
    c=a;a=b;b=c;
    printf("APRES: a=%d \t b=%d \n",a,b);
}

void permut2 (const int *a, const int *b)
{
    int *c;
    printf("AVANT: a=%d \t b=%d \n",*a,*b);
    *c=*a;
    *a=*b;
    *b=*c;
    printf("APRES: a=%d \t b=%d \n",*a,*b);
}
```

```
void permut2 (const int *a, const int *b)
```

```
main ()
{
    int u,v;
    u=2;v=3;
    permut1(u,v);
    printf("Permut1:\t u=%d \t v=%d \n",u,v);
    /*-----*/
    permut2(&u,&v);
    printf("Permut2:\t u=%d \t v=%d \n",u,v);
}
```

tableau et pointeur

```
#include <stdio.h>

void tmp( int *q, int val);
void tmp( int *q, int val)
{
    *q=val;
}

main()
{
    int v[10], *q;
    tmp(v,3);
    tmp(&v[1],4);
    q=v;
    printf("\nv[%d]=%d\n",0,*q);
    q++;
    printf("\nv[%d]=%d\n",1,*q);
}
```

Const & pointeur

- Les **pointeurs** peuvent être rendus **const**. Le compilateur s'efforcera toujours **d'éviter** l'allocation, le **stockage** et **remplacera** les expressions constantes par la **valeur appropriée** quand il aura affaire à des pointeurs **const**, mais ces caractéristiques **semblent moins** utiles dans ce cas. **Plus important**, le compilateur **vous signalera** si vous **essayez** de **modifier** un **pointeur const**, ce qui **améliore grandement** la **sécurité**.


```
int d = 1;  
const int* const x = &d; // (1)  
int const* const x2 = &d; // (2)
```

```
int* -Pointeur vers int  
int const * -pointeur const int  
int * const - const-pointeur int  
int const * const -const-pointeur const int
```

Maintenant la première const peut être de chaque côté du type :

- `const int * == int const *`
- `const int * const == int const * const`

const

```
const char *s; // read as "s is a pointer to a char that is constant"
```

```
char c;
```

```
char *const t = &c; // read as "t is a constant pointer to a "
```

```
char *s = 'A'; // Can't do because the char is constant
```

```
s++; // Can do because the pointer isn't constant
```

```
*t = 'A'; // Can do because the char isn't constant
```

```
t++; // Can't do because the pointer is constant
```

Next : Allocation Dynamique de la Mémoire

- E Allocation de mémoire
- E Déallocation de mémoire
- E Tableaux (n dimensions)
- E Arithmétique des pointeurs

Allocation de mémoire

Dans C comme dans C++, il y à 3 manières d'allouer de l'espace mémoire:

3 Mémoire Statique:

La mémoire est allouée par le **linker** au **début** du **programme**, et est libérée lorsque le programme à **fini d'exécuter**.

3 Mémoire Automatique:

La mémoire est automatiquement **allouée**, **gérée** et **libérée** pendant **l'exécution** du **programme**. Les **arguments** des **fonctions** et les **variables locales** obtiennent de **l'espace** de cette **manière**

3 Mémoire Dynamique:

La mémoire est **requis explicitement** par le programme(ur). Le programme(ur) **gère** et **libère** la memoire (en principe).

Où se trouve la variable?

compile-time program-text STATIQUE

- variables globales
- variables static

automatic stack pile AUTOMATIQUE

- variables locales
- parametres de fonctions
- valeur de retour des fonctions

run-time heap tas DYNAMIQUE

- malloc
- calloc
- Realloc

Les espaces mémoire alloués de manière statiques ou automatiques ne sont **généralement** pas 1 **problème** pour le **programmeur** (qui n'a généralement pas **besoin** de **s'en occuper**). Il faut cependant en être conscient pour pouvoir **optimiser** ces **programmes**.

```
int x; /* global */

int f(int n) {
    int x; /* local to f */
    if (n > 3) {
        int x; /* local to if */
        ... }
    {
        /* a local scope */

        int x;
    }
}
```

AUTOMATIQUE

Variables Locales

Paramètres de fonctions

Retours de fonctions

STATIQUE

Constantes

Variables globales

Variables déclarées: **static**

Allocation de mémoire dynamique :

- C** L'**allocation** de mémoire **dynamique** a par contre tendance à être 1 peu + **problématique** pour le **programmeur**. C lui qui **l'alloue**, qui la **gère** et qui n'oublie pas de la **rendre** au **système** quand il n'en a + **besoin**. Si la job est mal faite, attendez vous à des **problèmes!!!**
- C** Le heap sert à **l'allocation dynamique** de **blocs** de **mémoire** de **taille variable**.
- C** De **nombreuses structures** de **données** emploient tout **naturellement** l'allocation de **mémoire** dans le **heap**, comme par exemple les **arbres** et les **listes**.
- C** Le seul risque est la **fragmentation** du **heap**, par **allocation** et **libération** **successives**. Il n'existe pas en C de **mécanisme** de "ramasse-miettes" (**garbage collector**).

```
int * x = (int*)malloc(sizeof(int));  
int * a = (int*)calloc(10,sizeof(int));  
*x = 3;  
a[2] = 5;  
free(a);  
a = 0;  
free(x);  
x = 0;
```

Fonctions de gestion de mémoire (C)

```
void* malloc(size_t size);  
void* calloc(size_t n, size_t size);  
void* realloc(void * ptr, size_t size);  
void free(void * ptr);
```


Demande d'allocation de mémoire (**malloc**)

- *malloc* demande l'allocation d'un **bloc** de **mémoire** de *size* **octets** consécutifs dans la **zone** de **mémoire** du **heap**.
- **Syntaxe :**
 - `#include <stdlib.h>`
 - `void *malloc(size_t size);`
- **Valeur retournée :**
 - Si l'**allocation** réussit, *malloc* retourne un **pointeur** sur le **début** du bloc **alloué**. Si la **place disponible** est **insuffisante** ou si **size** vaut **0**, *malloc* retourne **NULL**.
- **Attention :** Les fonctions d'allocation dynamique **retournent** des **pointeurs** sur **des void**. Il faut donc opérer des **conversions** de **types explicites** pour **utiliser ces zones** mémoire en **fonction** du type des **données** qui y **seront mémorisées**.

```
#include <stdio.h>
#include <stdlib.h>
main() {
    char *ptr;
    ptr = (char *) malloc(80);
    /* demande d'allocation de 80 octets */

    if ( ptr == NULL)
    {printf("Allocation mémoire impossible\n"); exit(1);}

    /* libération de la mémoire */
    free(ptr);
}
```

Demande d'allocation de mémoire (**calloc**)

- La fonction *calloc* **réserve** un bloc de taille **nelemxelsize** octets **consécutifs**. Le bloc alloué est **initialisé** à **0**.
- **Syntaxe :**

```
#include <stdlib.h>  
void *calloc(size_t nelem, size_t elsize);
```
- **Valeur retournée :**
Si succès, **calloc** retourne un **pointeur** sur le **début** du **bloc alloué**. Si échec, *calloc* **retourne NULL** s'il n'y a plus **assez** de **place** ou si **nelem** ou **elsize** valent **0**.
- **Attention :** Les fonctions d'allocation dynamique retournent des **pointeurs** sur des **void**. Il faut donc **opérer** des **conversions** de **types explicites** pour utiliser ces zones **mémoire** en **fonction** du type des **données** qui y **seront mémorisées**.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int *str = NULL;
    str = (int *) calloc(10, sizeof(int));
    printf("%d\n", str[9]);
    free(str);
    return 0;
}
```

Demande d'allocation de mémoire (**realloc**)

La fonction *realloc* **ajuste** la **taille** d'un bloc à **size octets consécutifs**.

Syntaxe :

```
#include <stdlib.h>
```

```
void *realloc(void *ptr, size_t size);
```

Valeur retournée :

Si succès, retourne **l'adresse** de **début** du bloc **réalloué**.

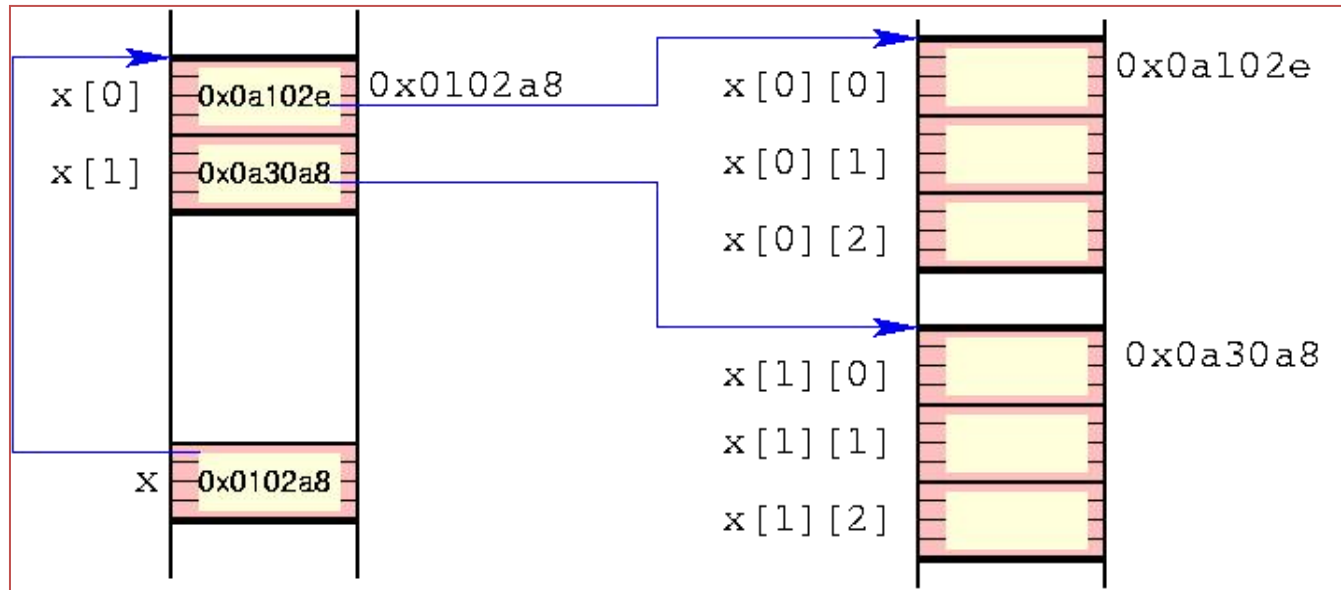
Cette adresse **peut** avoir **changé** par rapport à celle **fournie** en **argument**. Dans ce cas, le **contenu** de **l'ancien** bloc est **copié** à la **nouvelle adresse** et **l'ancienne zone** est **automatiquement libérée**. Si échec, (pas assez de place en mémoire ou *size* à 0), **realloc** retourne la **valeur NULL**

Arguments :

ptr : pointeur sur le début d'un bloc mémoire créé par

2 dimensions (matrices):

```
double ** alloc_matrix(int n, int m) {  
    double ** M = (double**) calloc(n, sizeof(double*));  
    int i;  
    for(i=0; i<n; ++i)  
        M[i] = (double*) calloc(m, sizeof(double));  
    return M;  
}
```



Libération

La fonction *free* libère un bloc mémoire d'adresse de début *ptr*.

Syntaxe :

```
#include <stdlib.h>  
void free(void *ptr);
```

Ce bloc mémoire a été précédemment alloué par une des fonctions *malloc*, *calloc*, ou *realloc*.

Attention :

Il n'y a pas de vérification de la validité de *ptr*. Ne pas utiliser le pointeur *ptr* après *free*, puisque la zone n'est plus réservée.

A tout appel de la fonction *malloc* (ou *calloc*) doit correspondre un et un seul appel à la fonction *free*.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    char *str;
    str = (char *) malloc (100 * sizeof (char));
    gets(str);
    /* saisie d'une chaine de caracteres */
    /* suppression des espaces en tete de chaine */
        while ( *str == ' ') str++;
    /* free ne libere pas toute la zone allouee
    car ptr ne designe plus le debut de la zone
    memoire allouee par malloc */

        free (str);
        return 0;
}
```


Déallocation de la mémoire de notre matrice 2 Dimensions:

```
void free_matrix( double** M, int n) {  
  
    int i;  
    for(i = 0; i<n; ++i)  
        free (M[i]);  
    free (M) ;  
  
}  
...  
free_matrix(M, 5) ;  
M = 0;
```

Arithmétique des pointeurs:

H Il est possible de **déplacer** la **position** d'un **pointeur** en lui **ajoutant** ou en lui **retranchant** une **valeur entière**. Par exemple:

```
ptr_str += 1;
```

Le compilateur avance d'un ou plusieurs octets par rapport à la position de **ptr_str**.

H Les entiers **ajoutés** ou **retranchés** ont des tailles **scalaires** **différentes** selon le **type** du **pointeur**. En d'autres termes, **ajouter** ou **soustraire 1** à un **pointeur** n'équivaut pas à se **déplacer** d'un octet, mais à aller à l'adresse suivante ou précédente.

Arithmétique des pointeurs:

Le format général du déplacement d'un pointeur est le suivant:

`pointeur + n`

n est un entier positif ou négatif. `pointeur` est le nom de la variable déclarée ainsi:

`type* pointeur;`

Le compilateur interprète l'expression `pointeur + n` comme suit:

`pointeur + n * sizeof(type)`

Arithmétique des pointeurs:

```
p[1][2] + 3 == (*(p + 1))[2] + 3
```

```
==
```

```
*(p[1] + 2) + 3 == (*(p + 1) + 2) + 3
```

Types dérivés

Prof. A.SABOUR

Objectifs du cours ...

Comprendre :

- Les Énumérateurs
- Les Structures
- Les Unions

Le type énumération : enum

C'est un type particulier à valeurs entières; à chaque énumération est associée un ensemble de constantes nommées :

```
enum jour_t {lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche};  
  
jour_t jour;  
...  
jour = lundi;
```

Par défaut, la valeur associée au **premier** nom est 0, au second 1, etc. Dans l'exemple précédent, la valeur associée à lundi sera 0, à mardi 1, etc. Il est possible de préciser la valeur associée à un ou plusieurs noms. Dans ce cas, les constantes qui suivent les constantes affectées sont **incrémentées de 1**.

```
enum mois_t={JAN=1, FEV, MAR, AVR, MAI, JUN, JUL, AOU, SEP, OCT, NOV, DEC};
```

JAN vaut 1, FEV 2, MAR 3, etc.

Le type énumération : enum

`enum` bool {FALSE,TRUE,FAUX=0,VRAI};

- Les identificateurs de la liste énumérée sont des constantes entières commençant par défaut à 0 et incrémentées
- La valeur entière d'un identificateur peut être spécifiée.
- Dans ce cas, les valeurs des identifiants suivants la prennent pour référence
- Ainsi : FALSE = 0, TRUE = 1 FAUX = 0 et VRAI = 1
- Donc : FALSE = FAUX = 0 et TRUE = VRAI = 1

Example

```
1 int main() {  
2     enum {sun, mon, tue, wed, thur, fri, sat} days;  
3     days = mon;  
4     if (days == mon)  
5     printf("I don't like Mondays\n");  
6 }
```

Technically:

sun is 0, mon is 1, tue is 2, ...

Therefore:

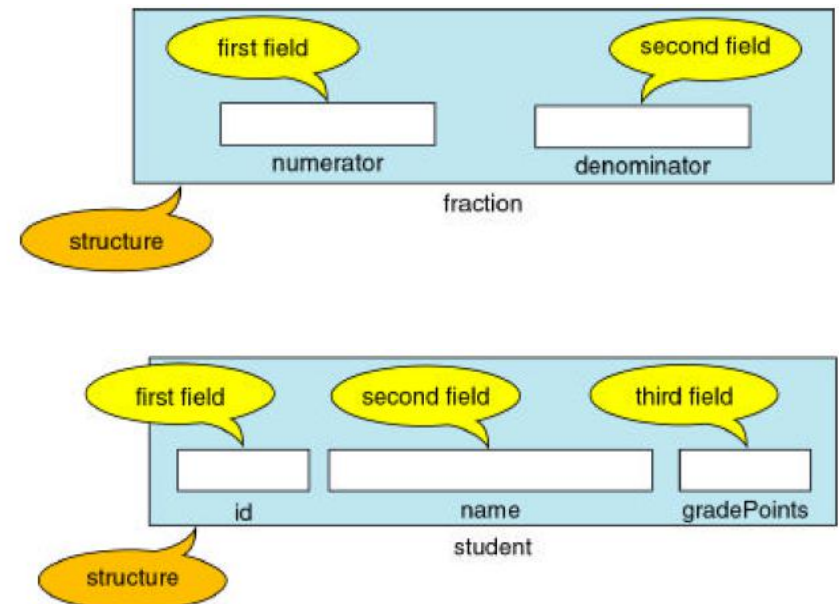
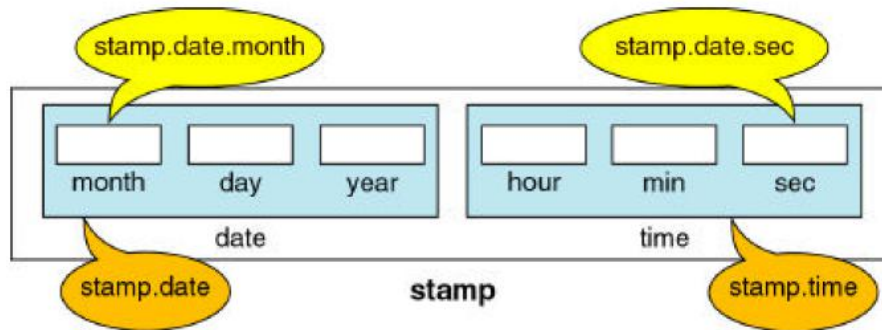
days = 1; /* is valid */

typedef && enum

```
1 enum DaysOfWeek {sun=1, mon, tue, wed, thur, fri, sat};
2 enum DaysOfWeek today;
3 enum DaysOfWeek yesterday;
4 // ou ecrire
5 typedef enum DaysOfWeek {sun=1, mon, tue, wed, thur, fri
, sat} DaysOfWeek;
6 DaysOfWeek today;
7 DaysOfWeek yesterday;
```

Les structures

- **Structure**
- La notion de structure permet de **manipuler** sous forme d'une **entité unique** un objet **composé d'éléments**, appelés **membres** ou **champs**, de types **pouvant être différents**.



Syntaxe

```
1  [typedef] struct [struct_name]
2  {
3      type attribute;
4      type attribute2;
5      // ...
6      [struct struct_name *struct_instance;]
7  } [struct_name_t] [struct_instance];
```

Initialisation

```
1  struct S { int a, b, c, d; };
2  struct S s = { 1, 2, 3, 4 };
3  /* 1 for `s.a`, 2 for `s.b` and so on... */
4  struct S s2 = { .c = 3 };
```

Déclaration d'une structure

- Voici un exemple de déclaration d'une structure:

```
struct personne {  
    char nom[20];  
    char prenom[20];  
    int no_ss;  
};
```

Initialisation d'une structure:

Voici un exemple d'initialisation d'une structure:

```
struct complexe {  
    double re;  
    double im;  
};  
struct complexe z = {1., 1.};
```

Accès aux champs d'une structure:

L'accès aux champs d'une structure se fait avec l'opérateur **.** (**point**). Par exemple, si l'on reprend la structure complexe z, on désignera le champ re par **z.re**. Les champs ainsi désignés peuvent être utilisés comme toute autre variable.

- Pour accéder au champ <truc> d'une variable structurée <identi_var>, il faut utiliser la syntaxe :
<identi_var>.<truc>
- C'est le "." qui permet d'accéder au champ à partir de l'identificateur <identi_var>.
- Si on accède au champ <truc> à partir d'un identificateur <identi_pt> de variable qui pointe vers une variable structurée, alors il faut utiliser la syntaxe:
<identi_pt> -> <truc>
- C'est le "->" qui permet d'accéder au champ à partir de l'identificateur <identi_pt>.

Pointeurs sur une structure

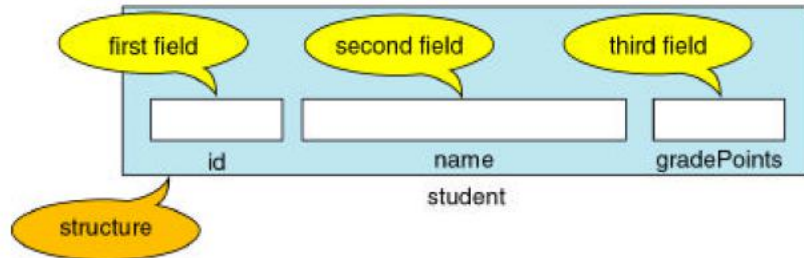
- L'accès aux champs d'une structure par l'intermédiaire d'un pointeur se fait avec l'opérateur '->' :

```
typedef char string[20];
struct personne {
    string nom;
    string prenom;
    int age; };
int main(){
    struct personne
    *p=malloc(sizeof(struct
    personne));
    strcpy(p->nom,"Dupont");
    strcpy(p->prenom,"Claude");
    p->age= 20;
    affichePtrPersonne(p);
}
```

Dans l'exemple, on aurait aussi pu accéder au champ age par: (*p).age

```
void affichePersonne( const struct personne A){
    printf("\n\t le nom : %s",A.nom);
    printf("\n\t le prenom : %s",A.prenom);
    printf("\n\t l'age : %d",A.age);
}
void affichePtrPersonne( const struct personne *
A){
    printf("\n\t le nom : %s",A->nom);
    printf("\n\t le prenom : %s",A->prenom);
    printf("\n\t l'age : %d",A->age);
}
```

Example



```
1 typedef struct {  
2     char id[10];  
3     char name[26];  
4     int gradePoints;  
5 } student;
```


Exercice A

```
typedef struct {  
    int hr;  
    int min;  
    int sec;  
} CLOCK;  
void increment(CLOCK *clock);  
void show(CLOCK *clock);  
CLOCK secTOclock(int A );  
int clockTOsec (const CLOC A);  
CLOCK diffToWclock (const CLOC A, const CLOC B);  
CLOCK addToWclock (const CLOC A, const CLOC B);
```

Les unions

- Une union est un objet qui contient, selon les **moments**, **l'un** de ses membres qui sont de types divers; une union permet de stocker à une même adresse mémoire des objets de types différents.

```
union etiq {  
    int x;  
    float y;  
    char c;  
};  
etiq u;
```

u aura une taille suffisante pour contenir un float (le plus grand des 3 types utilisés dans l'union).
A un instant donné u contiendra soit un entier, soit un réel, soit un caractère.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 int gett(const char c, const int p);
5 char sett(char c,int p,int val);
6 char coder(char c);

7 void tobin(const char c, char*t);
8 int gett(const char c, const int p){
9     if(p>=0 && p<8)
10         return ((c&(1<<p)) ? 1:0);
11     printf("Depassement du l'intervale"); exit(2);
12 }
```

```
13 char sett(char c,int p,int val){
14     if(val==0) c&=~(1<<p);
15     else c|=(1<<p);
16     return c;
17 }
```

```
18 char coder(char c){
19     char s;
20     int i,tp[8]={2,0,1,3,4,7,6,5};
21     for(i=0;i<8;i++)
22         s=sett(s,tp[i],gett(c,i));
23     return s;
24 }
```

```
30 int main()
31 {   int i,j; char c;
32     char *c=(char*)malloc(sizeof(char)*40);
33     printf("ENTREZ UNE chaine de CARACTERE\t");
34     gets(c);
35     for(i=0;i<strlen(c);i++){
36         printf("\n %c:%d:",c[i],c[i]);
37     return 0;
38 }
```

```
1 #include <stdio.h>
2 #include <stdlib.h>
```

```
3 typedef struct bito{
4     unsigned short
5     b0:1, b1:1, b2:1, b3:1,b4:1,
6     b5:1, b6:1, b7:1, b8:1,b9:7;
7 }bito;
```

```
8 typedef struct ubc {
9     union{
10     bito    b;
11     char    c;
12     };
13 }ubc ;
```

```
14 void affiche (ubc x);  
15 char echange (char v);
```

```
16 int main()  
17 {  
18 printf("Hello world !   %d \n",sizeof( unsigned short ));  
19 //     ubc u;  
20 //     u.c=6;  
21 //     affiche(u);  
22 int i;  
23 char S[50];  
24 printf("\n\t saisir une chaine : ");  
25 gets(S);  
26 printf("\n\t CHAINE CODER : ");  
27 //for(i=0;i<strlen(S), 2>1, 5<6;i++)  
28 for(i=0;i<strlen(S);i++)  
29     putc( echange(S[i]),stdout );  
30     return 0;  
31 }
```



```
32 void affiche (ubc x){
33 printf("\n\t %c:%d:%d:%d:%d:%d:%d:%d:",
34      x.c,x.b.b7,x.b.b6,x.b.b5,x.b.b4,x.b.b3,x.b.b2,x.b.b1,x.b.b0);
35 }
```

```
36 char echange (char v){
37 ubc t,x;
38 t.c=0;x.c=v;
39 t.b.b0=x.b.b1;
40 t.b.b1=x.b.b2;
41 t.b.b2=x.b.b0;
42 t.b.b3=x.b.b3;
43 t.b.b4=x.b.b4;
44 t.b.b5=x.b.b7;
45 t.b.b6=x.b.b6;
46 t.b.b7=x.b.b5;
47 return t.c;
48 }
```


Types dérivés

- En plus des types de base, il existe un nombre **théoriquement infini** de types pouvant être construits à partir des types de base :
- Des **tableaux** d'objets d'un certain type ;
- Des **fonctions** renvoyant des objets d'un certain type ;
- Des **pointeurs** sur des objets d'un certain type;
- Des **structures** contenant des objets de types divers ;
- Des **unions** qui peuvent contenir un objet parmi plusieurs de types divers.
- Ces constructions de types dérivés peuvent se faire en général de manière récursive.

Les Fichiers

Prof. A.SABOUR

Objectifs du cours ...

Objectif :

- La gestion de fichiers sous C

GENERALITES

- Un fichier est un **ensemble d'informations** stockées sur une **mémoire de masse** (disque dur, disquette, bande magnétique, CD-ROM).
- En C, un fichier est une **suite d'octets**.
- Les informations contenues dans le fichier ne sont pas **forcément** de même type (un char, un int, une structure ...)

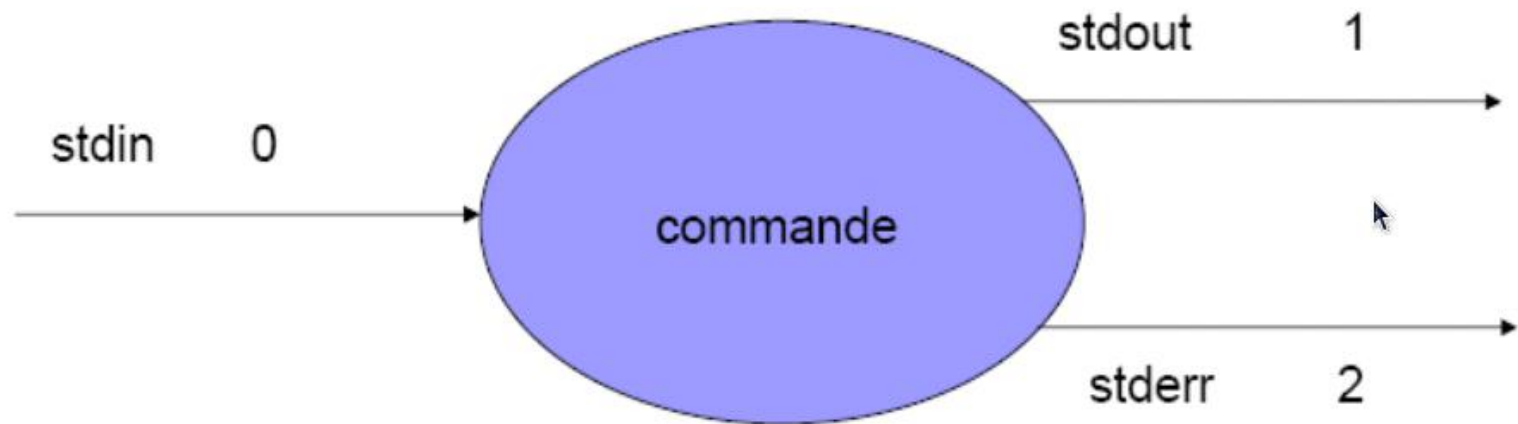
Les fichiers non standards

- En **réalité**, le clavier et l'écran sont des fichiers qui ont le nom **stdin** et **stdout**, respectivement.
- En C, nous avons la **possibilité** de définir nos propres fichiers dans lesquels nous pouvons lire et écrire nos informations.

Ces fichiers, à l'opposé du clavier et de l'écran, peuvent être **sauvegardés** et **consultés** ultérieurement.

Gestion des entrées/sorties

- A chaque commande est associée un canal d'entrée et deux canaux de sortie.



Par défaut: 0 = clavier 1 et 2 = écran

Exemple

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main()
4  {   printf("Hello world! 1 \n");
5      system("pause > file1");
6      fprintf(stdout,"Hello world! 2\n");
7      system("pause > file1");
8      fprintf(stderr,"Hello world! 3\n");
9      system("pause > file1");
10     return 0;
11 }
```

>"cours les fichier.exe" 1>ali.txt 2>baba.txt

FILE *stdin,*stdout,*stderr sont ouverts automatiquement au début du programme

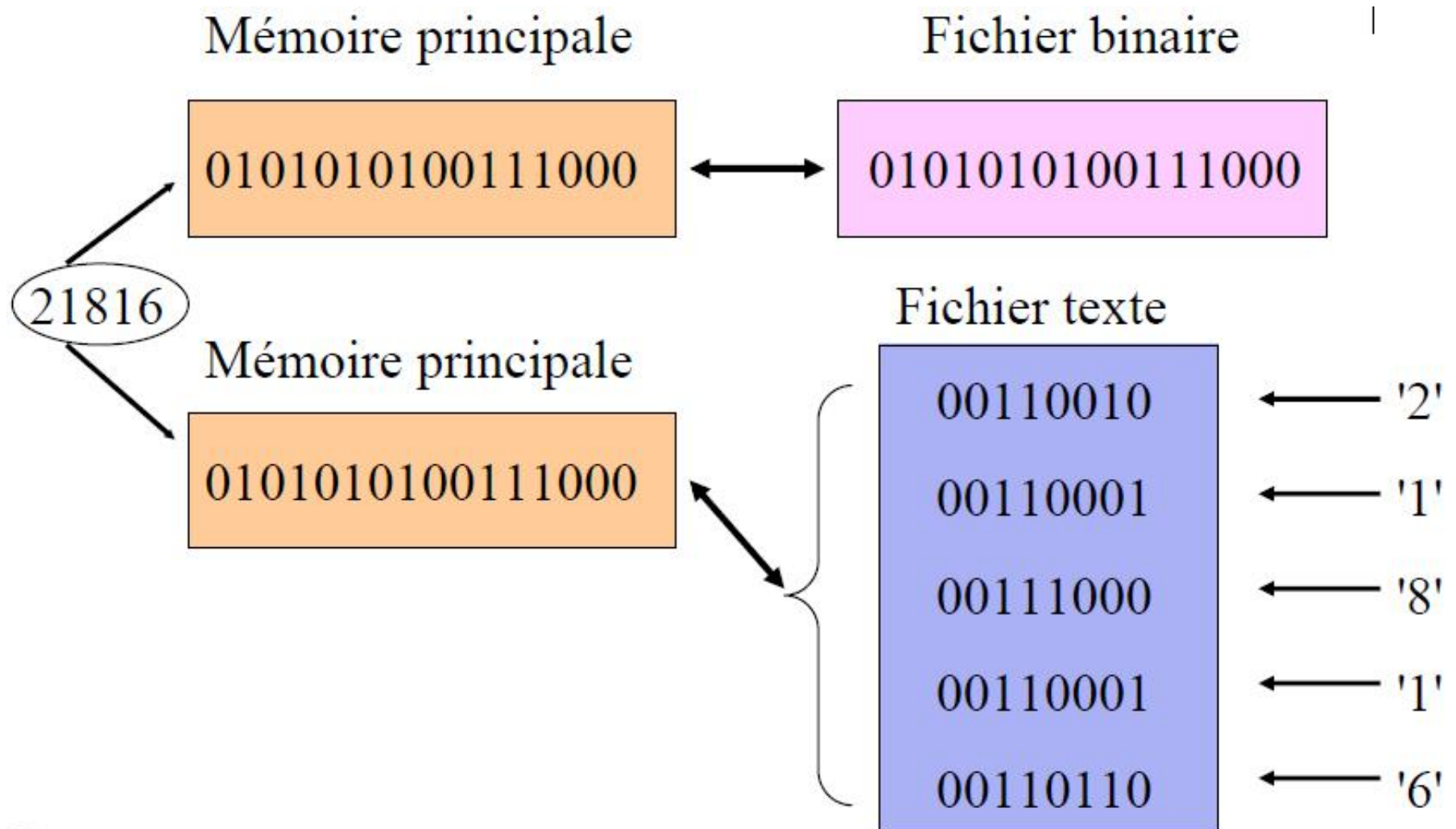
Fichiers textes

- Les fichiers textes sont des fichiers **séquentiels**.
- On écrit des codes **ASCII** de caractères uniquement.
- Fichier dit « texte », les informations sont codées en ASCII. Ces fichiers sont lisibles. Le dernier octet de ces fichiers est EOF (caractère ASCII spécifique).

Fichiers binaires

- Les données dans les fichiers sont représentées de la même façon que les données en mémoire (Complément à 2, IEEE754).
- L'accès aux données peut être **direct** par un **positionnement** à l'emplacement désiré. Il n'y a pas de **fin de ligne**.
- L'information est sous forme codée et généralement inintelligible lorsqu'elle est affichée avec un éditeur de texte.
- **Utilise généralement moins d'espace.**

Fichiers binaires vs fichiers textes



Le type FILE *

Pour pouvoir **travailler** avec un **fichier**, un programme a besoin d'un certain nombre **d'informations** au sujet du **fichier**:

- adresse de la **mémoire tampon**,
- **position** actuelle de la **tête** de lecture/écriture,
- **type d'accès** au fichier: écriture, lecture, ...
- état d'erreur**.

Ces informations sont rassemblées dans une **structure** du type spécial **FILE**. Lorsque nous **ouvrons** un fichier avec la commande **fopen**, le système génère automatiquement un bloc du type **FILE** et nous fournit

Le type FILE *

```
typedef struct _iobuf
{
    char* _ptr;
    int _cnt;
    char* _base;
    int _flag;
    int _file;
    int _charbuf;
    int _bufsiz;
    char* _tmpfname;
} FILE;
```

La lecture ou l'écriture dans un fichier n'est pas directe, mais utilise une zone mémoire tampon.

U n e s t r u c t u r e spécifique gère ce tampon et d'autre variables nécessaires à la gestion du processus.

Le type FILE *

```
1 void etatFILE(FILE *ptr) {
2     if(ptr!=NULL) {
3         printf(" \n les informations associées au fichier : ");
4         printf(" \n\t char*      _ptr : %s ",ptr->_ptr);
5         printf(" \n\t int      _cnt : %d ",ptr->_cnt);
6         printf(" \n\t char*      _base : %s ",ptr->_base);
7         printf(" \n\t int      _flag : %d ",ptr->_flag);
8         printf(" \n\t int      _file : %d ",ptr->_file);
9         printf(" \n\t int      _charbuf : %d ",ptr->_charbuf);
10        printf(" \n\t int      _bufsiz : %d ",ptr->_bufsiz);
11        printf(" \n\t char*      _tmpfname : %s ",ptr->_tmpfname);
12    }
13 }
```

Le type fichier

- On manipule les fichiers par l'intermédiaire de structures FILE décrites dans stdio.h

FILE *monFichier;

- Nom physique
 - Type d'accès(binaire ou formaté ... distinction floue)
 - Adresse du buffer
 - Position dans le fichier
 - Indicateur d'erreur
- Trois fichiers/flux standard, ouvert par défaut: stdin, stdout et stderr

Ouverture/Fermeture d'un fichier

- Faire le lien entre le nom physique (chaîne de caractère, chemin absolu ou relatif) et le nom logique (nom du pointeur)
- Ouvrir dans le mode choisi (binaire, écriture, écraser, lecture,...)
- Se positionner au début de fichier
- `fich = fopen(nom_physique, mode);`
TESTER si le fichier a été ouvert (pas null).
`fopen(nom, "r");` 0 si le fichier n'existe pas.
`fopen(nom, "w");` 0 si le chemin d'accès est faux.
`fclose(fich);`

Mode d'ouverture

- **r** ouverture d'un fichier texte en lecture
- **w** crée un fichier texte en écriture, écrase le contenu précédent si le fichier existait.
- **a** ajoute: ouvre ou crée un fichier texte et se positionne en écriture à la fin du fichier.
- **r+** ouvre un fichier en mode mise à jour (lecture et écriture)
- **w+** crée un fichier texte en mode mise à jour, écrase le contenu précédent si le fichier existait.
- **a+** ajoute, ouvre ou crée un fichier texte en mode mise à jour et se positionne en écriture à la fin du fichier.

Ouverture des fichiers

Les différents modes d'accès aux fichiers texte.

Mode d'accès	cible	résultat
"r"	Ouvrir le fichier en lecture (read)	fopen retourne NULL si le fichier n'existe pas
"w"	Ouvrir le fichier en écriture (write)	Le fichier est créé s'il n'existe pas. S'il existe le contenu est écrasé et perdu
"a"	Ajouter des données (append). Ouverture en écriture à la fin du fichier	Le fichier est créé s'il n'existe pas
"r+"	Ouvrir le fichier en lecture et en écriture	Fopen retourne NULL si le fichier n'existe pas
"w+"	Ouvrir le fichier en lecture et en écriture	Le fichier est créé s'il n'existe pas. S'il existe le contenu est écrasé et perdu
"a+"	Ouvrir le fichier en lecture et en ajout	Le fichier est créé s'il n'existe pas

Pour les fichiers binaires, les modes d'accès sont: "rb", "wb", "ab", "rb+", "wb+", "ab+".

Fermeture des fichiers

Quand un fichier n'est plus utilisé, on le ferme. Cela annule sa liaison avec le pointeur FILE correspondant.

Attention il ne faut pas oublier de fermer un fichier après utilisation, car le nombre de fichiers susceptibles d'être ouverts simultanément est limité (nombre de pointeurs FILE limité).

int fclose(FILE *pointeur_fichier);

Exemple:

```
1  #include <stdio.h>
2  void main(void) {
3      FILE *fp;
4      ...
5      ...
6      fclose(fp);
7      // Fermeture du fichier, retourne 0
8      // si pas d'erreur sinon EOF
9  }
```

exit (de n'importe où) et **return** dans **main** font automatiquement **fclose** de tous les fichiers ouverts.

Exemple

```
1 FILE *fich;  
2 char nom[100];  
3  
4 printf("nom du fichier à lire : \n");  
5 scanf("%s",nom);  
6 if( (fich=fopen(nom, "r")) ==NULL)  
7     exit(1);
```

Ouverture et fermeture de fichier

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

- nom=chemin d'accès au fichier
- mode=="r" pour ouvrir en lecture
- mode=="w" pour ouvrir en écriture (création ou remplacement)
- mode=="a" pour ouvrir en écriture à la fin du fichier
- Retourne NULL si erreur

`Int fclose(FILE *);`

- Ferme le fichier après un fflush éventuel
- Renvoie 0 ou EOF en cas d'erreur²

Lecture de fichiers

`char getc(FILE *);`

- lit un caractère et retourne sa valeur
- `getchar()` est équivalent à `getc(stdin)`
- renvoie EOF si fin de fichier ou erreur

`int fscanf(FILE *,const char *format,...);`

- `scanf(...)` est équivalent à `fscanf(stdin,...)`
- renvoie EOF si fin de fichier ou erreur

`char *fgets(char *line,int max,FILE *);`

- lit une ligne dans le buffer line de taille max
- renvoie NULL si fin de fichier ou erreur

`int feof(FILE *);`

- renvoie 1 si la fin de fichier est atteinte, 0 sinon

getc exemple

```
#include<stdio.h>

int main()
{
    char c;

    printf("Enter character: ");
    c = getc(stdin);
    printf("Character entered: ");
    putc(c, stdout);

    return(o);
}
```

Ecriture de fichiers

`int putc(int c, FILE *f);`

- Ecrit le caractère c
- renvoie sa valeur ou EOF si problème
- putchar(c) est équivalent à putc(c, stdout)

`int fprintf(FILE *, const char *format, ...);`

- printf(...) est équivalent à fprintf(stdout, ...)
- renvoie le nombre de caractères écrits ou EOF si erreur

Lecture et écriture formatées

Lecture :

char *fscanf(FILE *pointeur_fichier, char *chaine_formatee, variables,...,);

La fonction fscanf lit des données dans un fichier en les formatant. Elle retourne le nombre de données correctement lues si pas d'erreur. La valeur EOF signifie fin de fichier ou erreur.

```
1 long num;  
2 char nom[30];  
3 char prenom[30];  
4 fscanf(fp, "%ld %s %s", &num, nom, prenom);
```


Lecture et écriture formatées

Ecriture :

```
int * fprintf(FILE * pointeur_fichier, char  
*chaine_formatee, variables,...,);
```

La fonction fprintf écrit des données dans un fichier en les formatant, elle retourne le nombre de données correctement écrites si pas d'erreur, et EOF s'il y a une erreur.

```
1 fprintf( fp, "%ld %s %s", num, nom, prenom) ;
```

Lecture et écriture en bloc

Lecture :

int *fread(void *pointeur_Tampon,size_t taille,size_t nombre,FILE *point_fic);

La fonction fread lit un bloc de données de *taille* x *nombre* octets et le range à l'emplacement référencé par *pointeur_tampon*. Elle retourne le nombre d'octets lus. Si la valeur est inférieur à *nombre* alors erreur.

```
1 struct client k[5];  
2 fread(k, sizeof(struct client), 5, fp);
```

Lecture et écriture en bloc

Ecriture :

int * fwrite(void *pointeur_Tampon,size_t taille,size_t nombre,FILE *point_fic);

La fonction fwrite écrit un bloc de données de *taille* x *nombre* octets rangé à l'emplacement référencé par *pointeur_tampon* dans le fichier pointé par *point_fic*. Elle retourne le nombre d'objets complètement écrits. Si la valeur est inférieure à *nombre* alors erreur.

```
1 fwrite(k, sizeof(struct client), 5, fp);
```

Exercice

- Afficher à l'écran le contenu d'un fichier (idem commande **type**)

```
1 main() {
2     FILE * monfichier;
3     char sur_disque[100]; char c;
4     /* acquisition du nom */
5     scanf("%s",sur_disque);
6     /*ouverture*/
7     monfichier=fopen(sur_disque,"r");
8     if (monfichier==NULL) printf("erreur\n");
9     else { // lecture affichage
10         while ((c=getc(monfichier))!=EOF) printf("%c",c);
11         fclose (monfichier);
12     }
13 }
```

```

#include <stdio.h>
#include <stdlib.h>
/**
typedef struct _iobuf
{
    char*      _ptr;
    int        _cnt;
    char*      _base;
    int        _flag;
    int        _file;
    int        _charbuf;
    int        _bufsiz;
    char*      _tmpfname;
} FILE;

*/
void etatFILE(FILE *ptr) {
if(ptr!=NULL) {
    printf(" \n les informations associées au fichier : ");
    printf(" \n\t char*      _ptr : %s ",ptr->_ptr);
        // printf(" \n\t int        _cnt : %d ",ptr->_cnt);
        printf(" \n\t char*      _base : %s ",ptr->_base);
        // printf(" \n\t int        _flag : %d ",ptr->_flag);
        // printf(" \n\t int        _file : %d ",ptr->_file);
        // printf(" \n\t int        _charbuf : %d ",ptr->_charbuf);
        // printf(" \n\t int        _bufsiz : %d ",ptr->_bufsiz);
        // printf(" \n\t char*      _tmpfname : %s ",ptr->_tmpfname);
    }
}

```

```

char * readligne(FILE* fichier,char *ligne,const int Max);
char * readligne(FILE* fichier,char *ligne,const int Max ){
    int i=1,nb=0;
    char lettrenom=fgetc(fichier);
    printf("\n\n \t 97 44 66 ali baba ");
    etatFILE(fichier);
    while (lettrenom != '\n'&& i++<Max ){
        ligne[nb++]=lettrenom;
        lettrenom=fgetc(fichier);
        etatFILE(fichier);
    }
    if(i==Max) printf("Erreur d allocation ");
    ligne[nb]='\0';
    return ligne;
}

```

```

int main(int nbr,char **S)
{
    int i=0;

    FILE *ptr=fopen("file.txt","rt");
    int Max=255;
    char tmp[Max];
    tmp[255]='\0';
    while(!feof(ptr)){
        readligne(ptr,tmp,Max);
        printf("\n ligne %2d : %s",i++,tmp);
        // feof (FILE* __F) { return __F->_flag & _IOEOF; }
        //fscanf(ptr,"%[^'\n']",tmp);

        //fread(tmp,254,1,ptr);
        //etatFILE(ptr);
        system("pause >co");
    }

    fclose(ptr);
    return 0;
}

```