



Cours de programmation : langage C

Notes prises au cours de M.E. Wilfart

Ces notes n'ont pas été approuvées par le prof, elles ne peuvent donc être utilisés comme source sûre, des erreurs faites par l'auteur peuvent s'y trouver, Ceci n'est juste qu'un support pouvant éventuellement vous aider dans votre compréhension du langage C.

Dedeurwaerder Steve 1^{ère} informatique Don Bosco Tournai

Chapitre 1 : Structure d'un programme

Chapitre 2 : Les commentaires

Chapitre 3 : Les déclarations de variable

3.1 Introduction

3.2 Syntaxe

3.3 Les types de variables

3.3.1 type char

3.3.2 type int

3.3.3 type float

3.6 Les types structurés

3.6.2 Syntaxe

3.6.3 Syntaxe d'accès aux champs

3.8 Le type enum

3.8.1 Introduction

3.8.2 Syntaxe

3.9 Définition d'un nouveau type type def

3.9.1 Syntaxe

Chapitre 4 : Les déclarations des pointeurs

4.1 Introduction

4.2 Syntaxe

4.3 Initialisation des pointeurs

4.4 Utilisation des pointeurs

4.5 Réserve de la mémoire pour les pointeurs

Chapitre 5 : Les déclarations des tableaux

5.1 Introduction

5.2 Syntaxe

5.3 Initialisation d'un tableau

5.4 Allocation mémoire d'un tableau

5.6 Les tableaux et l'arithmétique des pointeurs

Chapitre 6 : les opérateurs

6.1 Introduction

6.2 Les opérateurs arithmétiques

6.2.1 L'opérateur d'addition +*

6.2.2 L'opérateur soustraction -

6.2.3 L'opérateur multiplication *

[6.2.4 L'opérateur division /](#)

[6.2.5 L'opérateur modulo %](#)

[6.3 Les opérateurs logiques](#)

[6.3.1 Introduction](#)

[6.3.2 l'opérateur ! \(de complément\)](#)

[6.3.3 L'opérateur ET &&](#)

[6.3.4 L'opérateur OU ||](#)

[6.4 Les opérateurs relationnels](#)

[6.5 Les opérateurs logiques « bit à bit »](#)

[6.5.1 opérateurs de complément ~](#)

[6.5.2 opérateur ET & \(bit à bit\)](#)

[6.5.3 Opérateur OU |](#)

[6.5.4 Opérateur OU exclusif ^](#)

[6.5.5 Opérateur de décalage >> , <<](#)

[6.6 Les opérateurs d'affectation](#)

[6.6.1 affectation simple =](#)

[6.6.2 affectation arithmétique](#)

[6.6.3 affectation binaire](#)

[6.6.4 incrémentation et décrémentation unaire](#)

[6.6.5 L'opérateur sizeof](#)

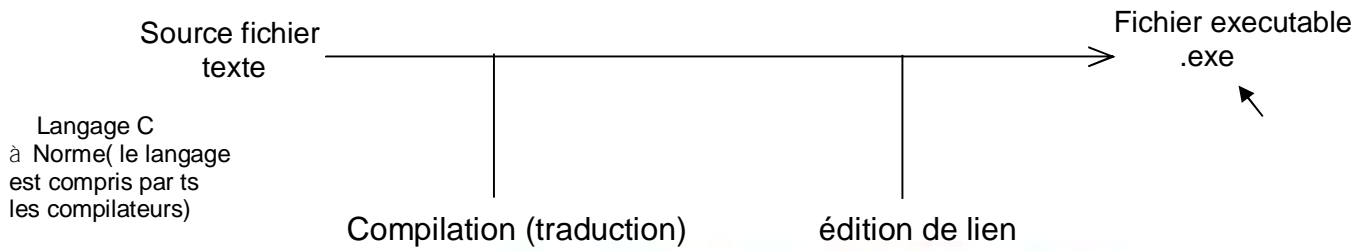
[Chapitre 8 Les instructions en C](#)

[8.1 Introduction](#)

[8.2 Instruction de contrôle](#)

[8.3 L'instruction de boucle](#)

Chapitre1 : Structure d'un programme



Dans le fichier source, on retrouve plusieurs parties :

```
#include <stdio.h>
typedef struct {
    char nom [20] ;
    char prénom [20] ;
} fiche ;
void fonction() ;
int a ;
void main ()
{
    printf (« bonjour à tous \n ») ;
}

void fonction ()
{
    printf (« fonction\n ») ;
}
```

à 1.directive de précompilation

à 2. définition des types

à 3. déclaration des prototypes des fonctions

à 4. déclaration des variables

à 5. définition des fonctions

Programme simple :

```
#include <stdio.h>
void main ()
{
    printf("bonjour à tous\n");
}
```

- 1) attention à la majuscule != minuscule
ex : main () != Main()
- 2) Toute ligne de code doit se terminer par un ;
- 3) Attention aux blocs d'instructions

Chapitre 2 : Les commentaires

Les commentaires peuvent se mettre en blocs sur plusieurs lignes.

```
/* β début de commentaire  
*/ β fin de commentaire
```

ex :

```
/* programme réalisé le 26 septembre 2003  
par adolf dubois  
affiche : bonjour à tous */  
  
# include <stdio.h>  
void main ()  
    {  
        printf ("bonjour à tous\n");  
    }  
/* fin du programme */ à ex1
```

Chapitre 3 : Les déclarations de variable

3.1 Introduction

Une variable est un emplacement mémoire que l'on peut faire varier

1) il faut fournir un type :

- ∅ int (entier)
- ∅ float } (données réelles)
- ∅ double
- ∅ char (caractère)

2) il faut donner un identificateur (nom de la variable)

Ex : `int a ;` à la variable s'appelant <a> et capable de contenir un entier

3) Eventuellement initialiser la variable (c-à-d : mettre quelque chose dans cette case mémoire)

- a) soit initialiser la variable lors de la déclaration
- b) soit initialiser la variable après la déclaration

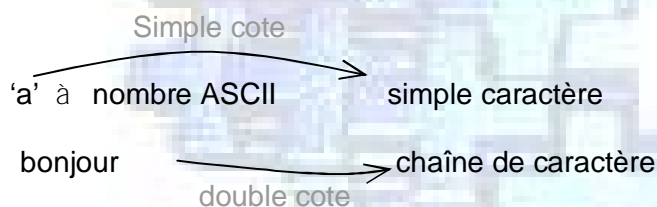
ex :

a) `int a=10 ;` opérateur d'affectation

`float prix = 10,3 ;`

b) `int a ;`
`a=20 ;` à l'initialisation peut se faire n'importe où

Remarque pour les variables de type char :



R\ : `char a = bonjour`
à `char caractere = 'a' ;`

`printf (code ascii du caractère %d\n ,caractere) ;` à [ex 2](#)
à affiche le contenu en entier

En C, les variables peuvent être créés :

1) Avant la fonction main

Exemple :
`#include <stdio.h>`
`int compteur ;`
`void main`
{
:
}

2) Dans une fonction MAIS avant toute ligne de code

Exemple :

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

```
void main()
```

```
{
```

OK à `int x ;`

```
printf (« bonjour »);
```

~~`int y ;`~~ NON

```
}
```

Dans les cases mémoires on ne retrouve que des entiers :

Code ASCII sur 8 bits

Entier positif (8bits)
 (0000000) 0 à 255 (1111111) (26minuscules, 26 majuscules +ponctuation)

Entier positif (16 bits)
 (0.....0) 0 à 65535 (1.....1)

car en C type char à 8 bits
 type int à 16 bits(short)
 à 32 bits(long)

ex :

```
unsigned short int a ;
a=65535 ;
a=a+2 ;
printf (« valeur en code ascii :%d\n »,a) ; à ex3
```

à variable non signé où le bit de poids fort n'est pas utilisé pour le signe, impossible de représenter des valeurs négatives codé sur 16 bits

ce code va afficher 1 car :

ex

```
toto=255            1 1 1 1 1 1 1 1
                  +
toto=toto+2        0 0 0 0 0 1 0
                  +-----+
                  1 0 0 0 0 0 0 1
```

```
char toto=0 ;
toto=127 ;
printf (« valeur encode ascii %d\n »,toto) ;
toto=toto+10;
printf (« valeur encode ascii %d\n »,toto) ; à ex4
```

à affiche -119

signé -128 à 0 à 127

```
127            0 1 1 1 1 1 1 1
+1            +
              0 0 0 0 0 0 1
              +-----+
              1 0 0 0 0 0 0    à -128
```

on doit prendre le complément restreint + 1

3.2 Syntaxe

[permanence] [classe de stockage] [modifieur] type identificateur [= valeur ;]
facultatif

à les modifieurs : unsigned
signed (par défaut)
long
short

à les types : char
int
float
double
void

3.3 Les types de variables

3.3.1 type char

Une variable déclarée de ce type peut contenir le code ASCII d'un caractère.
Le code ASCII permet d'associer une lettre de l'alphabet à un chiffre.
1 code ASCII est codé sur 8 bits

Nombre de représentation avec 8 bits : 2^8 (256) à 1 bit 0 1 2^1
2 bits 0 0 } $4 = 2^2$
0 1 }
1 0 }
1 1 }

ex : char test ='a' ; à la variable contient en réalité l'entier 97

3.3.2 type int

Une variable de ce type peut contenir un entier. Cette variable est codée sur 16 bits
Supposons que la variable soit non signée :

Valeur minimale : 0
Valeur maximale : 65535 } 2^{16} possibilités

Ex :

```
#include <stdio.h>

void main ()
{
    unsigned int compteur ;
        for (compteur =0; compteur<67000;compteur++)
            {
                printf(« %d\n »,compteur) ;
            }
}
```

à [ex5](#)

à nous avons dans ce cas une boucle sans fin car nous avons déclaré un entier court non signé nous avons 2^{16} possibilités, et pas une de plus

3.3.3 type float

Stockage de nombres réels en virgule flottante sur 4 octets avec une précision de 7 chiffres.

$3,4^e-38$ à $3,4^e+38$

(rappel 1 octet = 8bits)

`long float test ;` `long float ó double`



nombre réel code sur 8 octets avec une précision de 15 chiffres

$1,7^e-308$ à $1,7^e+308$

ex:

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

```
void main ()
```

```
{            à le résultat est 3 car 'result' est déclaré comme un entier
```

```
int result;
```

```
result = 10/3;
```

```
printf("%d\n",result);
```

```
}            à ex6
```

3.6 Les types structurés

Struct

Caractère à char

Entier à int

Réel simple précision à float

Réel double précision à double

type structuré à struct

3.6.2 Syntaxe

Struct client

```
{
```

```
    int référence ;
```

```
    float dettes ;
```

```
    int classe ;
```

```
};
```

exercice :

- 1) donner la syntaxe pour réserver un réel simple précision
- 2) donner la syntaxe pour réserver un entier court non signé.
- 3) Soit une structure appelé client devant contenir 2 entiers x et y.
 - a) comment déclarer la structure
 - b) comment déclarer une variable par rapport à cette structure
 - c) comment s'appellent en fait les variables x et y ?

- 1) { float x ; }
- 2) { unsigned short int x ; }

- 3) a) Struct client

```
{  
    int x ;  
    int y ;  
};
```

- b) Struct client a ;

- d) x et y sont des champs

3.6.3 Syntaxe d'accès aux champs

Struct clients

```
{  
int x ;  
int y ;  
};
```

struct clients a ;

void main()

```
{  
a.x = 10 ;  
a.y = 20 ;
```

nom de la variable . nom du champ

nom du pointeur à nom du champ

*

exemple :

() champ de bit

Struct decoupe

```
{  
unsigned int part 1 :4 ;  
unsigned int part 2 :8 ;  
unsigned int part 3 :4 ;  
};
```

int main (int argc, char*argv[])

```
{  
unsigned short int a=20237;  
struct decoupe * b;  
b=(struct decoupe*) &a; // transtipage à le * précise que c'est un transtipage d'adresse  
printf ("champ1:%d\n", b -> part 1);  
printf ("champ2:%d\n", b -> part 2);  
printf ("champ3:%d\n", b -> part 3);
```

system ("pause");

return 0;

}

à [ex7](#)

à [ex7bis](#)

à ce prog permet de decouper un nombre en 4,8,4 bit

(20237)₁₀

à (0 1 , , .. 1)₂

4 240 13

3.8 Le type enum

3.8.1 Introduction

Le type énuméré reprend sous une même entité une liste ordonnée d'identificateurs qui sont initialisés lors de la création d'une variable de ce type.

3.8.2 Syntaxe

a) déclaration du type énuméré

b) création des variables d'un type énuméré donné

a) enum identificateur

```
{  
  identificateur 1,  
  identificateur 2,  
  identificateur 3  
}
```

b) enum identificateur ident var ;

△ à la structure

ex :

```
enum qualité  
{  
  mauvais ;    β associe une variable entière à 0  
  bon ;        à 1  
  très bon ;   à 2  
  pp          β pas de virgule  
};
```

```
int main (int argc, char*argv[ ])  
{  
  enum qualité tilleul ;  
  tilleul = bon ;  
  if(tilleul >= bon)  
    printf (« récompense\n ») ;  
  else  
    printf (« pendu dans le fond des toilettes ») ;  
  system (« pause ») ;  
  return 0 ;  
}
```

à [ex8](#)

ex :

```
enum qualité
{
    mauvais ;
    bon ;
    très bon ;
    pp
};
```

Dans ce cas, trois variables entières sont créés :

- 1) mauvais qui contient 0 ó int mauvais =0 ;
- 2) bon qui contient 1 ó int bon =1 ;
- 3) très bon qui contient 2 ó int très bon =2 ;

si on veut commencer la liste avec une valeur # 0, on affecte une valeur au 1^{er}.

Ex : mauvais =10 à les variables qui suivent seront incrémenté de 1 à chaque fois

() les variables Booléennes

```
enum BOOL à le type booléen natif n'existe pas
{
    FALSE, à 0
    TRUE à 1
};
```

```
int main ()
{
    int a= 10 ;
    while (a)
    {
        printf ("valeur a:%d\n",a);
        a=a-1 ;
    }
    system (« pause ») ;
    return 0 ;
} à ex9
```

3.9 Définition d'un nouveau type type def

3.9.1 Syntaxe

```
typedef type identificateur ;
```

Ex:

```
UINT a ; unsigned short int
```

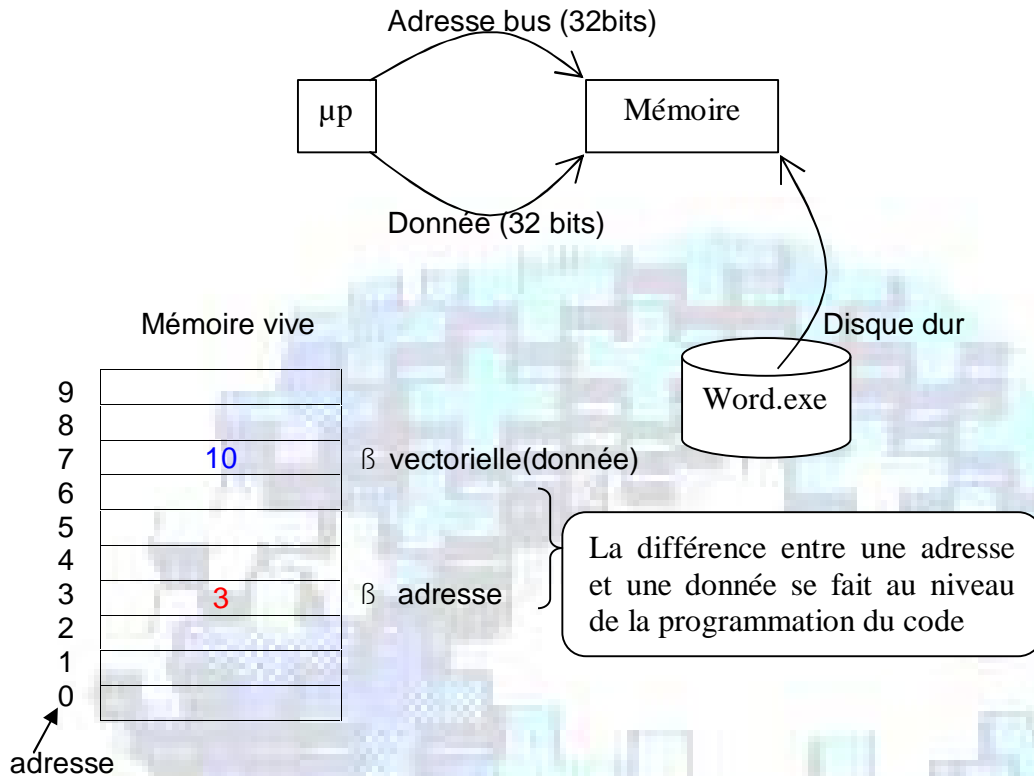
à déclaration du type
typedef unsigned short int UINT;

à [ex10](#)

Chapitre 4 : Les déclarations des pointeurs

4.1 Introduction

Un pointeur est un emplacement mémoire qui va contenir l'adresse d'une autre case mémoire



4.2 Syntaxe

`int a ;` β déclaration d'une variable
`int *b ;` β déclaration d'un pointeur
 (pour l'instant il n'y a rien dedans à il faut l'initialiser)

4.3 Initialisation des pointeurs

! Tout pointeur doit être initialisé avant usage !

2 techniques :

1. écrire directement une adresse connue dans le pointeur
ex : `int*b ;`
 `b=0X08000000 ;`
2. utiliser l'adresse d'une variable existante
`int a ;`
`int *b ;`
`a=10 ;`
`b=&a ;` à on met l'adresse de a dans b

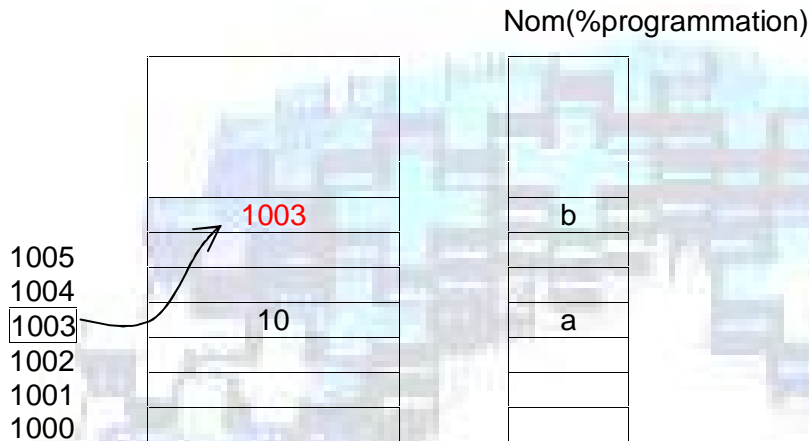
& à opérateur : adresse de

Lors de la déclaration d'un pointeur, il faut renseigner le type de la variable dont l'adresse y est contenue

Si on déclare un entier (pointeur)
Le type de l'adresse qui contient doit être un entier

Ex :

```
Float x ;
Float *y ;
X=10.SF ;
Y=&X ;
```



4.4 Utilisation des pointeurs

1. utilisation directe du contenu du pointeur qui est une adresse
2. accès au contenu de la case mémoire dont l'adresse est dans le pointeur à (c'est ce qu'on appelle de la redirection)

ex :

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

```
void main ()
```

```
{
  int *pointeur;
```

(pointeur d'entier code sur 4 octet ou 2 si short)

```
int a;
```

```
int*b;
```

à ici * : déclaration du pointeur

```
a=10;
```

```
b=&a;
```

```
printf("contenu de a:%d\n",*b);
```

à ici * : redirection (ou indirection)

```
printf (« taille de a :%d »,size of(a)) ;
```

```
printf (« emplacement du pointeur :%d\n »,&pointeur) ;
```

```
system (« pause ») ;
```

```
return 0 ;
```

```
}
```

à [ex11](#)

ex :

```
int main()
{
    int a=10 ;
    float b=20.0f ;
    int *pointeur ;
    pointeur=&a ;
    ptrfloat=&b ;

    printf(« taille de a :%d »,sizeof(a)) ;
    printf(« taille du pointeur d'entier :%d »,sizeof(pointeur)) ;
    printf(« taille du pointeur de flottant :%d »,sizeof(ptrfloat)) ;
    printf(« adresse de a :%d »,&a) ;
    printf(« adresse de b :%d »,&b) ;

    system(« pause ») ;
    return 0 ;
}
```

à [ex12](#)

4.5 Réserve de la mémoire pour les pointeurs

Mémoire vive (640 Ko)



) 64 Ko à un segment

Pour accéder à une case mémoire il faut :

1. adresse de segment (16 bits)
2. déplacement dans le segment (offset) à 16 bits

Lorsque l'on veut accéder à une case mémoire présente dans le segment actif, il suffit de fournir l'offset.

à 2 types d'adresses : adresses courtes (16bits) à renseigne uniquement l'offset
 adresses longues (32 bits) à renseigne l'offset et le segment

à ensemble des modèles mémoires :

	Segments de code	Segments de données
TINY	1 seul segment (<64k>)	
SMALL	1 segment	1 segment
MEDIUM	1 segment	1 segment
COMPACT	1 segment	Plusieurs (tableau <64k)
LARGE	Plusieurs segments de code	Plusieurs segments de données (tableau < 64 k)
HUGE	Plusieurs segments de code	Plusieurs segments de données (tableau > 64 k)

à Les types de pointeurs :

- pointeurs courts (16bits)
- NEAR (small, medium)
- pointeurs longs (32bits)
- FAR (compact, large)
- HUGE (huge)

La taille d'un pointeur dépend du modèle mémoire et du type.

à Tableau à 2 dimensions

```
Ex int tab [2] [3] = { {1,4,7} , {6,2,1} } ;  
int tab [ ] [ ] = { {1,2,6} , {7,4,5} } ;
```

/* Remarque sur les chaînes de caractères :*/

- En C, il n'y a pas de type permettant de gérer une chaîne de caractère
- Une chaîne de caractère en C, c'est un tableau de type char

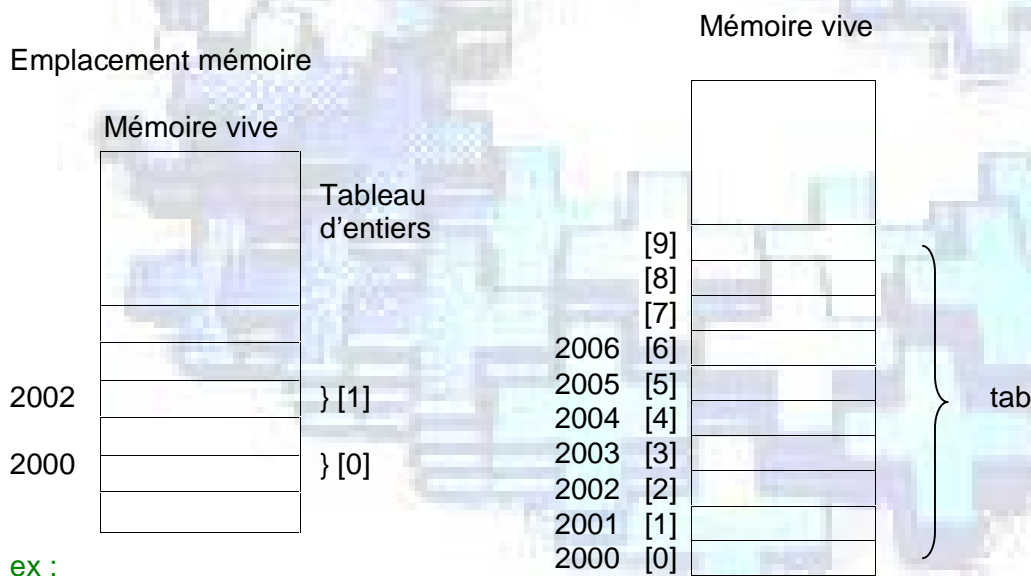
Ex : char nom [50] ;

- L'initialisation peut aussi s'effectuer lors de la déclaration

```
Ex : char nom [ ] = {'b','o','n',' ','j','o','u','r'} ;  
char nom [ ] = {« bonjour »} ;  
char liste [3] [10] = {« un », «deux », «trois »} ;  
= { {'u','n'}, {'d','e','u','x'} } ;
```

5.4 Allocation mémoire d'un tableau

Ex char tab [10] ;

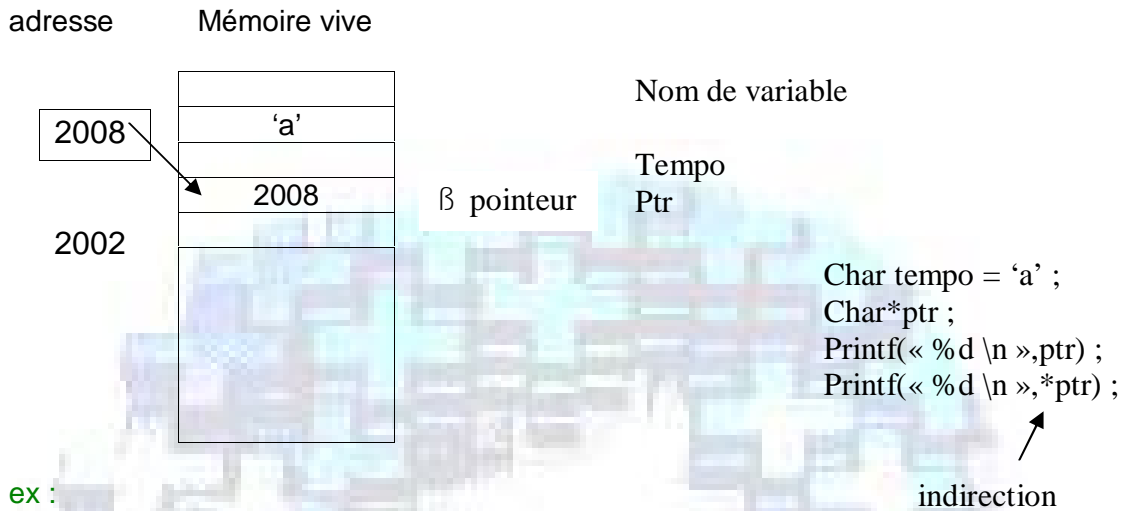


```
int main (int argc, char *argv[ ] )  
{  
char tab [ ] = (« bonjour ») ;  
int i ;  
  
printf (« adresse de tab : %p\n », &tab) ;  
printf (« contenu de tab : %x\n », tab) ;  
printf (« indirection de tab [0] : %c\n », *tab) ;  
printf (« contenu de tab [0] : %c\n », tab[0]) ;  
printf (« adresse de tab [0] : %p\n », &tab[0]) ;  
  
system (« pause ») ;  
return 0 ;  
}
```

à [ex13](#)

! Le nom d'un tableau est en fait le nom d'un pointeur dont le contenu est une adresse pointant sur le premier élément du tableau !

() l'indirection



ex :

```
int main (int argc, char*argv[ ])
{
int tab [ ] = { 1,2,3,4,5,6,...,16 } ;
int*ptr ;
```

```
ptr = &tab[4];
printf(« %d\n »,ptr[2]) ; à va afficher 7 pq ? parce que prt = adresse de tab [4]
```

01234 à va commencer au 5^e élément ptr[2]=+2 =7

```
systeme (« pause ») ;
return 0 ;
} à ex14
```

à Tableau à plusieurs dimensions

ex :

```
int main(int argc, char*argv[ ])
{
char tab [3] [5]= (« un », »deux », »trois ») ;
printf(« %C\n »,tab [1] [1]) ; à affiche e
printf(« %C\n »,ptr [4]) ;
System (« pause ») ;
Return 0 ;
}
```

à ex15

U	N		D	E	U	X		T	R	O	I	S
[0][0]	[0][1]	[0][2]	[1][0]	[1][1]	[1][2].....							

Ex :

```
int main()
{
char nom [50];
char prenom [50]=("Lucie") ;
char*adresse= "TOURNAI" ;

strcpy (nom,"Dupond");

printf("nom :%s\n",&nom[2]);          //affiche pond
printf("prénom :%s\n",prenom);
printf("adresse :%s\n",adresse);

system ("PAUSE") ;
return 0 ;
} à ex16
```

Une chaîne de caractère correctement terminée l'est toujours par le caractère NUL à '\0'

Ex :

```
int main ()
{
char nom [50] ;
char*adresse = " TOURNAI " ;
char*ptr=&nom[5] ;
strcpy(nom,"bonjour à tous") ;
printf (" %C\n",ptr [6]) ;

system("PAUSE") ;
return 0 ;
} à ex17
```

1. Affichage du printf ?

à affiche o

B	O	N	J	O	U	R		A		T	O	U	S	\0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
					0	1	2	3	4	5	6			

2. Quelle est la taille en octet du tableau associé à « adresse » (octet) ?
Toute chaîne de caractère se termine par un caractère Nul

à char*adresse = »TOURNAI » ;

↓
8 octets

car T O U R N A I \0 (ne pas oublier de compter le caractère NUL)
1 2 3 4 5 6 7 8

- Concaténation de 2 chaînes de caractères

```

Ex : int main()
    {
      char part1 [50] ;
      char part2 [50] ;
      strcpy (part1, "bonjour") ;
      strcpy (part2, "a tous") ;
      part1[3]='\0';           //caractère nul à la 3éme lettre bon

      printf ("%s\n",part1) ;
      printf ("%s\n",&part1[4]) ;
      printf ("%s\n",part2);

      system("PAUSE");
      return 0;
    }
  
```

à [ex18](#)

mais il existe une fonction pour concaténer les chaînes de caractères

```

à STRCAT(part1,part2) ;
   printf (« « %S\n »,part1) ;
  
```

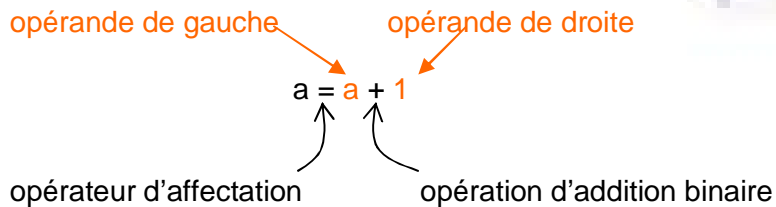
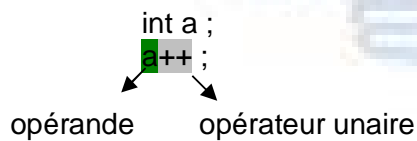
à [ex19](#)

5.6 Les tableaux et l'arithmétique des pointeurs

2 opérateurs : + , -

3 types d'opérateurs :
 opérateur unaire à 1 opérande
 opérateur binaire à 2 opérandes
 opérateur ternaire à 3 opérandes

ex :



Dans le cadre des pointeurs la 2éme opérande ne peut être qu'un entier.

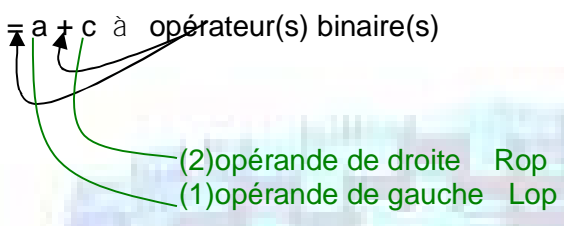
Chapitre 6 : les opérateurs

6.1 Introduction

Classification des opérateurs en fonction du nombre d'opérandes qu'ils doivent traiter

Ex : a) `a++` à opérande unaire
Opérateur

b) `b = a + c` à opérateur(s) binaire(s)



Dans le cas de l'opérateur '`+`', celui-ci retourne un résultat

c) `x = a ? b : c`

équivalent niveau code

```
if (a)
{
x=b ;
}
else
{
x=c ;
}
```

à [ex23](#)

ex :

```
int main()
{
int a =10 ;
int b= 3 ;
float c=(float)a/b;
float d=a/b;
printf ("résultat:%f\n",c);
printf("résultat sans le transtipage:%f\n",d);

system("PAUSE");
return 0;
}
```

transtipage

à [ex20](#)

!!! pour le compilateur

```
int / int = int
float / int = float
int / float = float
```

Représentation d'un flottant `float=3.0F` ; à signifie que ceci est un flottant et pas un double.

Les différents opérateurs :

Les opérateurs arithmétiques (+ , - , / , * , %)

Les opérateurs logiques (&&, ||, !)

! à complément de

Les opérateurs conditionnels (== , > , < , <= , >= , !=)

Les opérateurs d'affectations (= , += , -= , *= , &=)

Les opérateurs logiques bit à bit (masquages, décalage) (&, | , ^ , ~ , >> , << , ? :)

Les opérateurs d'accès aux champs d'une structure (. , - >)

Les opérateurs d'adressage (&, *)

() Il n'y a pas d'opérateurs pour les puissances (a^b) ou exclusif
à mais il existe des fonctions

`a += b`

`ó`

`a = a + b`

y a-t-il une différence ?

non, c'est la même chose niveau fonctionnement (mais pas niveau rapidité)

`a++` à 1opérateur unaire

`a = a + 1` à 2opérateurs

6.2 Les opérateurs arithmétiques

6.2.1 L'opérateur d'addition +

Lop	Rop	Resultat
Pointeur	Entier	Adresse
Entier	Entier	Entier
Entier	Float	Float
Float	Float	Foat

Voir opération arithmétiques
sur les pointeurs

```
int main(int argc, char*argv[ ])
{
float a[50];
float*b;
b=(a+3);
printf("adresse de base %p\n",a);
printf("adresse %p\n",b);

system("PAUSE");
return 0;
}
```

à [ex21](#)

6.2.2 L'opérateur soustraction -

Lop	Rop	Resultat
Pointeur	Entier	Adresse
Entier	Entier	Entier
Entier	Float	Float
Float	Float	Float
Float	Entier	Float

6.2.3 L'opérateur multiplication *

Lop	Rop	Resultat
Entier	Entier	Entier
Entier	Float	Float
Float	Float	Float

6.2.4 L'opérateur division /

Lop	Rop	Resultat
Entier	Entier	Entier
Entier	Float	Float
Float	Entier	Float
Float	Float	Float

*

```
int a = 10 ;
```

```
int b = 3 ; // ! c contient 3 comme réponse
```

```
float d = a / b ; // à ex20
```

6.2.5 L'opérateur modulo %

Lop	Rop	Resultat
Entier	Entier	Entier

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

```
int main (int argc, char*argv[ ])
{
int a=10 ;
int b=40 ;
printf(« 10 modulo 40 %d \n »,a%b) ;
system("PAUSE");
return 0;
}
à ex22
```

6.3 Les opérateurs logiques

6.3.1 Introduction

Il n'existe pas de variables booléennes en c !

On utilise des entiers

- valeur 0 à FAUX
- valeur # 0 à VRAI

Toutes les instructions et tous les opérateurs logiques devant travailler avec des variables booléennes » travaillent en fait avec des entiers

Ex :

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

```
int main(int argc, char*argv[])  
{
```

```
int i=20 ;
```

```
while (i)
```

```
{
```

```
printf (« valeur de i : %d\n »,i) ;
```

```
i=i-1 ;
```

```
}
```

```
system (« PAUSE ») ;
```

```
return 0 ;
```

```
}
```

à [ex24](#)

while(...) ó faire tant que
la condition est vrai

Ex :

Voir feuille photocopier

6.3.2 l'opérateur ! (de complément)

A	A !
Vrai	Faux
Faux	Vrai

Ex :

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

```
int main (int argc, char* argv[ ])
{
  int i=20 ;
  int f= !20 ;
  int j= !f ;
  printf(« contenu de j : %d\n »,j) ;
  system(« PAUSE ») ;
  return 0 ;
}
```

à [ex25](#)

6.3.3 L'opérateur ET &&

A	B	A&&B
Faux	Faux	Faux
Faux	Vrai	Faux
Vrai	Faux	Faux
Vrai	Vrai	Vrai

Ex :

```
{
  int a=0 ;
  int b=0 ;
  int reponse = a&& b ;
  printf (« contenu de reponse :%d\n »,reponse) ;
  system(« PAUSE ») ;
  return 0 ;
}
```

à [ex26](#)

6.3.4 L'opérateur OU ||

A	B	A B
Faux	Faux	Faux
Faux	Vrai	Vrai
Vrai	Faux	Vrai
Vrai	Vrai	Vrai

Ex :

```
{  
int a=10 ;  
int b=10 ;  
int reponse = a||b ;  
printf (« contenu de reponse :%d\n »,reponse) ;  
system (« PAUSE ») ;  
return 0 ;  
}
```

à [ex27](#)

6.4 Les opérateurs relationnels

<	infériorité	opérandes de tout type mais les opérateurs retournent un entier qui sera fonction de la comparaison : FAUX à 0 VRAI à #0
<=	infériorité large	
>	supériorité	
>=	supériorité large	
==	égalité	
!=	inégalité	

Ex :

```
Int main (int argc, char*argv[ ])  
{  
char buffer='a' ;  
char buffer2='b' ;  
if (buffer !=buffer2) } Ce sont les codes qui sont comparés  
{  
printf ("chaines différentes\n");  
}  
else  
{  
printf (« chaines identiques\n) ;  
}  
system (« PAUSE ») ;  
return 0 ;  
}
```

à [ex28](#)

6.5 Les opérateurs logiques « bit à bit »

6.5.1 opérateurs de complement ~

ex :

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

```
int main (int argc, char*argv[ ])  
{  
  unsigned char a=127 ;  
  unsigned char b=~a;  
  printf("contenu de b:%x\n",b); // %x affichage en hexadécimal  
  system(« PAUSE ») ;  
  return 0 ;  
}
```

à [ex29](#)

	128	64	32	16		8	4	2	1
(ici127)aà	0	1	1	1		1	1	1	1
~aà	1	0	0	0		0	0	0	0
hexadécimal	8					0			

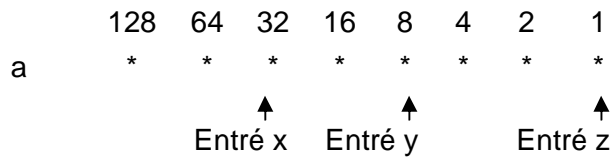
6.5.2 opérateur ET & (bit à bit)

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

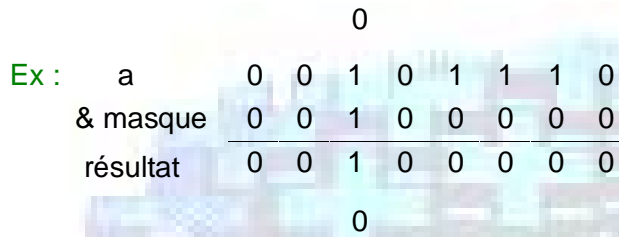
```
int main (int argc,char*argv[ ] )  
{  
  char a=127;  
  char b= 79;  
  printf("contenu de b:%d\n",a&b);  
  system(« PAUSE »)  
  return 0 ;  
} à ex30
```

a(127)	0	1	1	1	1	1	1	1	a(119)	0	1	1	1	0	1	1	1
b(72)	0	1	0	0	1	0	0	0	b(72)	0	1	0	0	1	0	0	0
a&b	0	1	0	0	1	0	0	0	a&b	0	1	0	0	0	0	0	0
a&b=72									a&b=64								

Soit un registre d'état dans un composant électronique et le contenu de ce registre est placé dans une variable de type « char » non signé dans votre programme.

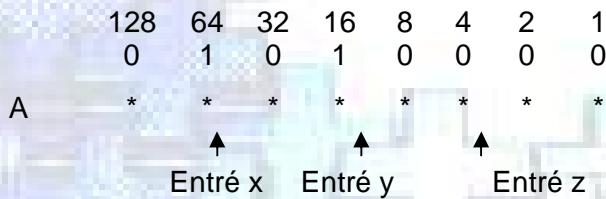


Dans le composant il y a 3 entrées logiques

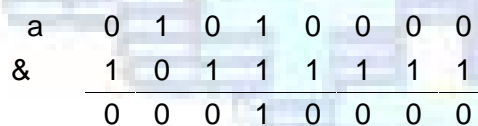


si le résultat vaut 0, l'entrée x était à zéro
 " " est # 0, l'entrée x était à "1"

L'opérateur & est appelé opérateur de masquage



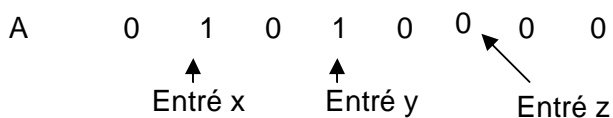
Ex : soit positionner la sortie x à 0 en laissant les autres sorties inchangées



à Positionnement d'un bit à 0

6.5.3 Opérateur OU |

à Positionnement d'un bit à 1



Ex : soit positionner la sortie z à 1 en laissant les autres sorties inchangées

a	0	1	0	1	0	0	0	0
	0	0	0	0	0	1	0	0
	0	1	0	1	0	1	0	0

6.5.4 Opérateur OU exclusif ^

Permet de complémenter un bit en particulier sans modifier les autres

A	0	1	0	1	0	0	0	0
		↑		↑		↑		
		Entré x		Entré y		Entré z		

Ex : on désire complémenter la sortie x et la sortie z sans modifier les autres

a	0	1	0	1	0	0	0	0
^	0	1	0	0	0	1	0	0
	0	0	0	1	0	1	0	0

a	b	a^b
0	0	0
0	1	1
1	0	1
1	1	0

6.5.5 Opérateur de décalage >> , <<

32	16	8	4	2	1
année de référence 2000					
			jour à 5 bits		
			mois à 4 bits		
			année à 7 bits		
			16bits(2octets)		

16 bits	*	*	*	*	*	*	*	*	*	*	*	*	*
	année				mois				jour				
	(% à 2000)												

```

int a ;
a=42788 ;

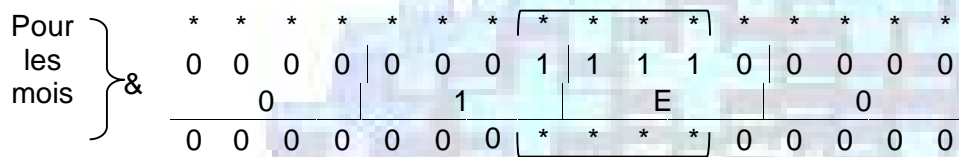
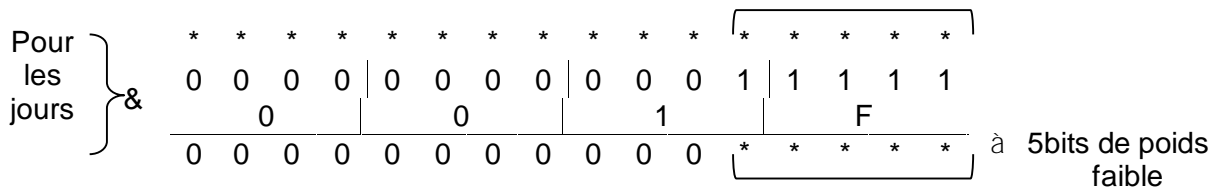
#include <stdio.h>
int main (int argc, char*argv[ ])
{
    unsigned short int a=42788;
    int jour =a&0x001f;           //a c'est un masque (0xà en hexadécimal)
    int mois=a&0x01e0 ;
    mois=mois>>5 ;               // on décale de 5
    int annee=(a>>9)+2000 ;      // on décale de 9

    printf (« jour :%d\n »,jour) ;
    printf (« mois :%d\n »,mois) ;
    printf (« année :%d\n »,annee) ;

    system(« PAUSE ») ;
    return 0 ;
}
    
```

à [ex31](#)

masque
unsigned a



Pour les années, il faudrait décaler de 9 positions mais sans pour autant faire un masque

Ex :

Compter le nombre de bits à '1' dans un entier

Void main ()

```
{
unsigned long int a=61276 ;
```

a β 1 1 1 0 1 1 1 1 0 1 0 1 1 1 0 0

si a%2 1) donne 0 à nombre est pair et le bit e de poids faible vaut 0
2) # de 0 à nombre est impaire et le bit de poids faible est à 1

void main ()

```
{
int b=0; //compteur de bit à 1 initialisé à 0
unsigned long int a= 61276 ; //condition initial
while (a!=0) //condition du traitement
{
if ((a%2) !=0) //tant que le reste de la division est != 0
b=b+1 ; //on incrémente b
a=a>>1 ; // et on décale a d'un bit
}
printf (« nbr de bits à 1 :%d\n »,b) ;
```

à [ex32](#)

ex :
voir feuille photocopié

aussi...

```
while (a)
{
if (a%2)
b=b+1 ;
a=a>>1;
}
```

aussi...

```
while (a)
{
b+=a%2;
a=a>>1;
}
```

Un opérateur de décalage est plus rapide qu'un opérateur arithmétique

```
While(a)
{
b=b+(a%2)
a=a/2
}
```

autre solution

à on décale le masque

&	1	1	1	0	1	1	1	1	0	1	0	1	1	1	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
&	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
&	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

```
Void main ()
{
int b=0 ;
unsigned long int a=61276 ;
int m=1;
while(m!=0)
{
if((a&m) !=0)b=b+1 ;
m=m<<1 ;
}
```

à [ex33](#)

6.6 Les opérateurs d'affectation

6.6.1 affectation simple =

6.6.2 affectation arithmétique

```
ex : a=a+b ; ó a+=b ;
      *          *=
      /          /=
      -          -=
      %          %=
```

6.6.3 affectation binaire

ex : $a=a<<1$; ó $a<<=1$;
 &
 |
 ^
 >>
 &=
 |=
 ^=
 >>=

6.6.4 incrémentation et décrémentation unaire

$a=a+1$ ó $a++$;
 $a=a-1$ ó $a--$; } on retrouve la pré incrémentation et la post-incrémentation

ex:

```
int main (int argc, char* argv[ ])
{
int b;
int c;
int a=10;
b=a++;           //post incrémentation
printf("a=%d\n",a);
a=10;
c=++a;          //pré incrémentation
printf("b=%d\n",b);
printf("c=%d\n",c);
system("PAUSE");
return 0;
}
```

à [ex34](#)

(affiche a=11 a=11 b=10 c=11)

ex:

```
int main(int argc, char* argv[ ])
{
int a;
float b=10.2F;
a=b;
printf("%d\n",a);
system("PAUSE");
return 0;
}
```

à [ex35](#)

transtipage:

$a=(int)b$ pour ne pas avoir de warning disant qu'on risqué de perdre des données

ex :

```
int a ;
double x=10,2 ;
float b=10,4F ;
a=(int)b ;
b=(float)x;
printf("voici a %d\n",a);
printf("voici b %f\n",b);
system(« PAUSE ») ;
return 0 ;
```

à [ex36](#)

Remarque

-Post incrémentation:
On affecte puis on incrémente

-Pré incrémentation:
On incrémente puis on affecte

6.6.5 L'opérateur sizeof

Opérateur permettant de renseigner la taille (en octet) d'un type ou d'une variable

Ex :

```
#include<stdio>

{
int main(int argc,char*argv[ ])
printf(« taille d'un entier :%d\n »sizeof(int)) ;
printf(« taille de a : %d\n »sizeof(a)) ;
system(« PAUSE ») ;
return 0 ;
}
```

à [ex37](#)

à avec structure

```
struct client
{
char nom [25] ;
char prenom [25] ;
short int classe ;
long int reference ;
}
```

```
printf(« taille de le structure client :%d\n »,sizeof(struct client)) ;
```

à [ex38](#)

! attention !
la taille d'une structure est géré en fonction des critères d'alignements mémoire (doit être divisible par 8 sinon c'est "ajusté")

Chapitre 8 Les instructions en C

8.1 Introduction

On peut classer les instructions du langage C en trois grandes familles:

1. Les instructions de base
Expression
Instruction nulle
blocs d'instructions
2. Les instructions de contrôle
if...else
switch ...case
3. Les instructions de boucle
while
do...while
for
4. Les instructions d'échappement
break
continue
return
goto

8.2 Instruction de contrôle

à if....else

à switch.....case

a) l'instruction if

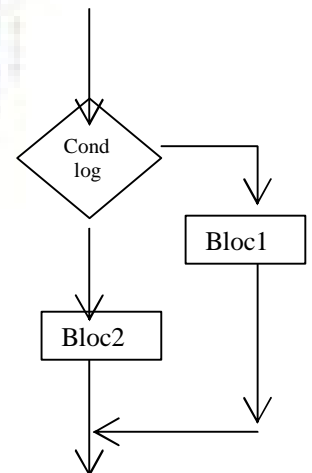
syntaxe :

```
if(condition logique) {  
    bloc d'instruction 1  
}
```

R : pas { } si une seul instruction

```
else  
{  
    bloc d'instruction 2  
}
```

R : facultatif



à c'est n'importe quel entier à valeur 0 : faux à bloc 2
à valeur#0 :vrai à bloc1

ex :

```
int main ()
{
int a = 10 ;
if(a)                                à va afficher le if si a#0   else si a=0
{
printf("condition vrai \n");
}
else
{
printf("condition fausse\n");
}
systeme(« pause ») ;
return 0 ;
}                                     à ex39
```

ex ;

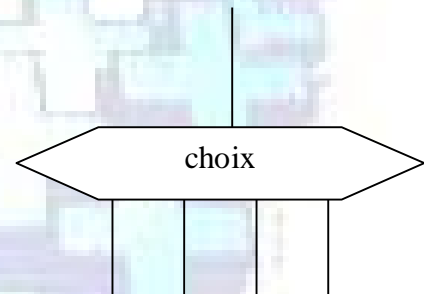
```
char*nom= »test » ;
if(nom == « test »)
{
.
.
.
}                                     à ex40 (à terminer)
```

on ne peut pas utiliser cet opérateur pour comparer une chaîne de caractère, il faut utiliser une fonction strcmp

b) L'instruction switch ... case

Syntaxe

```
Switch(expression)
{
[case expression constante]
[instruction]
[default:
instruction ]
}
```



ex:

```
{
int a=1
switch(a)                            à expression entière
{
case 1:
printf ("val de a :1\n");
.
.
case 3:
printf (".....");
default:
printf("Valeur non reprise\n");
}
system(« PAUSE.... »)                à ex41
```

Le break est une instruction d'échappement ici dans l'exemple si on ne met pas de break il exécute tout ce qui se trouve après l'élément choisi.

à il peut être intéressant de modifier l'ordre des « cases » car il contrôle toutes les cases jusqu'au bon.

R\ une variable simple de type char est considérée comme un entier codé sur un seul octet

8.3 L'instruction de boucle

à For

à While

à do...while

a) while

Syntaxe

```
While(condition logique)
{
bloc d'instructions
}
```

ex :

```
int main ()
{
int a = 10 ;
while (a)
{
printf("valeur a:%d\n",a);
a = a - 1 ;
}
system("pause");
return 0;
}
```

à [ex42](#)

b) do....while

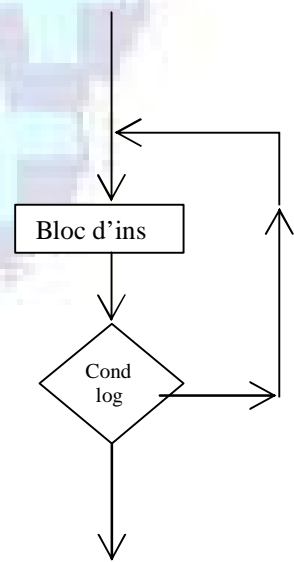
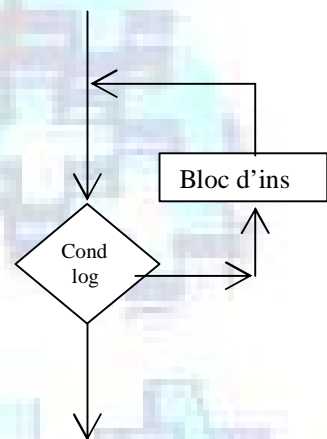
Syntaxe

```
Do
{
bloc d'instructions
}
while(condition logique) ;
```

Ex:

```
{
int a=10;
do
{
printf("valeur: %d\n",a);
a--;
}
while(a);
printf("valeur de a après la boucle %d\n",a);
system("PAUSE");
return 0;
}
```

à [ex43](#)

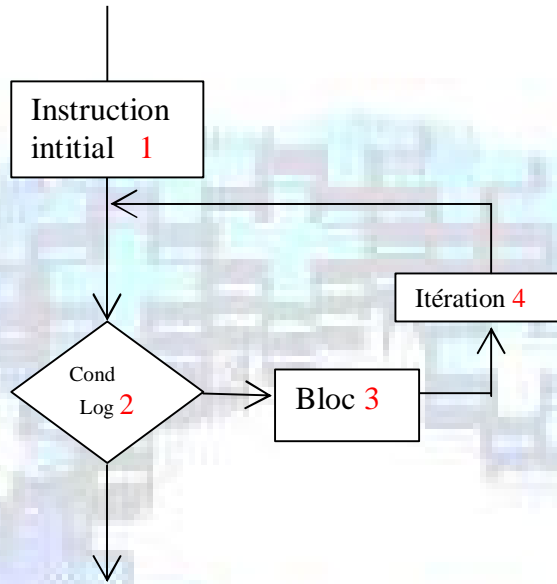


c) for

Syntaxe

```
For ( .....;.....;.....)
{
  bloc d'instruction
}
```

instruction initial
condition logique
Itération



Ex :

IDEM (au niveau du résultat)

Avec 'for'

```
Int main( )
{
  int i ;
  for (i=10 ;i>=0 ;i--)
  {
    printf("valeur de i :%d\n", i);
  }
  system("pause");
  return 0;
}
à ex44
```

Avec 'while'

```
Int main( )
{
  int i ;
  i=10 ;
  while(i>=0)
  {
    printf("valeur de i:%d\n", i);
    i--;
  }
}
à ex45
```

R/ dans un for l'instruction, la condition, l'itération ne dépend pas les unes des autres

Exercice 1

Soit une chaîne de caractère. Prévoir le code permettant de compter le nombre de majuscules et le nombre de minuscules

```
int main ( )  
{  
char x ;  
printf(« %d\n »,x) ;
```

Code ASCII

A :65 à Z :90

a : 97 à z : 122

```
system("pause");  
return 0;  
}  
à ex46
```

```
char*tab »Bonjour à Tous »\0
```

extremité de la chaîne

compteur de majuscules
compteur de minuscules

```
int i,min=0,maj=0 ;  
for(i=0 ;tab[i] !='\0';i++)  
{  
if((tab[i]>="a")&&(tab[i]<="z"))min++;  
if((tab[i]>="A")&&(tab[i]<="Z"))maj++;
```

```
#include <stdio.h>  
int main(int argc,char*argv[ ] )  
{  
char*tab="Bonjour à Tous les amis";  
char tab [ ]= »coucou » ;  
int maj=0, min = 0  
int i= 0  
  
while(tab[i] !='\0')  
{  
if((tab[i]>=97)&&(tab[i]<=122))min++;  
if((tab[i]>=65)&&(tab[i]<=90))maj++;  
i++;  
}  
printf("nombre de maj : %d",maj);  
printf(« nombre de min : % d »,min) ;  
system(« PAUSE ») ;  
return 0 ;  
}
```

à [ex47](#)

Exercice 2

Trie par ordre croissant

```
Char*tab1= »zut »  
Char*tab2= »wilfart » ;  
  
Int i=0;  
While ((tab1[i]==tab2[i])&&(tab1[i]!='0'))  
i++;  
If ((tab1[i]!='0')&&(tab2[i]!='0'))  
{  
printf("chaines identiques\n");  
}  
else  
{  
if(tab1[i]<tab2[i])  
printf(« chaine %s première \n »,tab1) ;  
else  
printf("chaine%spremière\n",tab2);  
}  
system(« PAUSE ») ;  
return 0 ;  
}
```

à [ex48](#)

Exercice 3

Inversion d'une chaîne de caractère

' b o n j o u r ' strlen(tab) à compte les caractères

char*tab= »bonjour » ;

' r u o j n o b '

```
#include<string.h>  
int taille ;  
char*tab= »bonjour » ;  
taille=strlen(tab) ;
```

```
printf(»d\n »,taille) ;  
int strlen (char*buffer)  
{  
:  
:  
}
```

à terminer.... à [ex49](#)