

la programmation en langage C

Dr Mahamadi BOULOU
Enseignant à l'Université Norbert ZONGO

Objectifs

- Être capable de bien programmer
- Comprendre les différentes constructions de la programmation en C
- Savoir programmer de manière modulaire

Qualités attendues d'un programme

- Clarté
- Simplicité
- Efficacité
- Modularité
- Extensibilité

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

TYPE de la valeur de retour

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

int main(void)

begin {
/* corps du programme */
declaration des Cstes et Var ;
instruction1 ;
instruction2 ;
....
}
end

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 (0 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 C

```
#include <stdio.h>
#define DEBUT -10
#define FIN 10
#define MSG "Programme de démonstration\n"
```

Directives du préprocesseur :
accès avant la compilation

```
int fonc1(int x);
int fonc2(int x);
```

Déclaration des fonctions

```
void main()
```

```
{
    int i;
```

/ début du bloc de la fonction main */*
/ définition des variables locales */*

```
    i = 0 ;
    fonc1(i) ;
    fonc2(i) ;
```

Programme principal

```
}
```

/ fin du bloc de la fonction main */*

```
int fonc1(int x) {
    return x;
}
```

Définitions des fonctions

```
int fonc2(int x) {
    return (x * x);
}
```

Indenter = lisibilité

Prenez l'habitude de respecter (au moins au début) les règles :

- une accolade est seule sur sa ligne,
- { est alignée sur le caractère de gauche de la ligne précédente,
- } est alignée avec l'accolade ouvrante correspondante,
- après { , on commence à écrire deux caractères plus à droite.

Fonctionnement :

- Taper et sauvegarder le programme,
- Compiler le programme,
- Exécuter le programme.

```
#include <Lib1.h>

#include <Lib2.h>

#define X 0;

int fonc1(int x);
float fonc2(char a);


int main(void)
{ /*main*/
    instruction;
    instruction;
    {
        instruction;
        {
            instruction;
        }
    }
    instruction;
} /* fin main*/
```

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**)

Les fonctions mathématiques (**math.h**)

Taille des type entiers (**limits.h**)

Limites des type réels (**float.h**)

Traitement de chaînes de caractères (**string.h**)

Le traitement de caractères (**ctype.h**)

Utilitaires généraux (**stdlib.h**)

Date et heure (**time.h**)

/* Entrees-sorties standard */

1^{er} Programme

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

```
#include <conio.h>
```

```
int main(void)
```

```
{
```

```
printf(" BTS GI ");
```

```
getch() ;    /* Attente d'une saisie clavier */
```

```
return 0;    /* En principe un code d'erreur nul signifie "pas d'erreur". */
```

```
}
```

BTS GI

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 "
- \r : retour chariot

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

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

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

caractères	nom	symbole	code hexa	décimal
\n	newline	LF	10	
\t	tabulation	HT	9	
\b	backspace	BS	8	
\r	return	CR		13
\f	form feed	FF	12	
\\	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 [argument, ...]);`

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 */



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.

1^{er} Programme

```
#include <stdio.h>
#include <conio.h>
int main(void)
{
    int age;          /*déclaration d'une variable*/

    printf("Je te souhaite le bon"
          "jour aux TP\nEt je t'"
          "e souhaite bon trav"
          "ail\n");

    printf("Quel est ton âge? ");
    scanf(" %d", &age);    /* lecture de l'âge, on donne l'adresse de age */
    printf("\nAlors ton age est de %d ans!\n",age);

    getch() ;           /* Attente d'une saisie clavier */
    return 0;          /* En principe un code d'erreur nul signifie "pas d'erreur". */
}
```

L'utilisation de **&** est indispensable avec **scanf** (valeur lue et donc modifiée), pas avec **printf** (valeur écrite et donc non modifiée).



Quel est ton âge ? **18**

Alors ton age est de **18** ans!

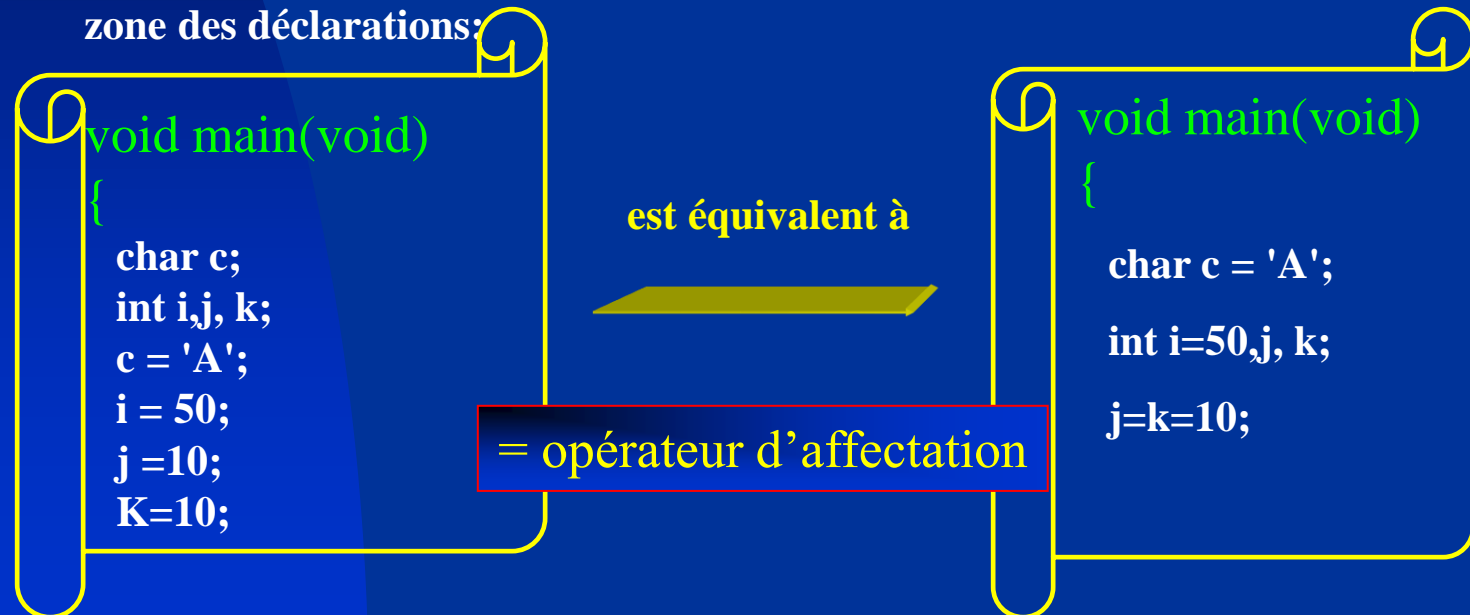
Variables : déclarations

Syntaxe : Type identificateur1, identificateur2, ..., ;

Exemple: char c1, c2, c3;
int i, j, var_ent;

Variables : initialisations

Les variables doivent être déclarées avant leur utilisation dans un début de bloc (juste après {}),



Cette règle s'applique à tous : char, int, float ...





scanf 

Exemples	entrées	résultats
<pre>char c1, c2, c3; scanf(''%c%c%c'',&c1,&c2,&c3);</pre>	a b c	c1=a c2=<espace> c3=b
<pre>scanf('' %c %c %c'',&c1,&c2,&c3);</pre>		c1=a c2=b c3=c
<pre>char c; int i; float x; scanf(''%2d %5f %c'',&i,&x,&c);</pre>	12 123.567 r	i=12 x=123.5 c=6

Affichages et saisies

Librairie : stdio.h

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

LES DECLARATIONS DE CONSTANTES

1ere méthode: définition d'un symbole à l'aide de la **directive** de compilation **#define**.

Exemple: **#define PI 3.14159**
void main()
{
 float perimetre, rayon = 8.7;
 perimetre = 2*rayon*PI;

}

Le compilateur ne réserve pas de place en mémoire

Syntaxe : **#define identificateur** texte(valeur)



#define TAILLE 100
#define MAXI (3 * TAILLE + 5)
#define nom_macro(identif_p1 , ...) texte
#define SOMME(X,Y) X+Y
#define MULTIP(A,B) (A)*(B)



Les identificateurs s'écrivent traditionnellement en majuscules, mais ce n'est pas une obligation.

LES DECLARATIONS DE CONSTANTES

2eme methode: **déclaration** d'une variable, dont la valeur sera constante pour tout le programme.

Exemple:

```
void main()  
{  
  const float PI = 3.14159;  
  const int JOURS = 5;  
  float perimetre, rayon = 8.7;  
  perimetre = 2*rayon*PI;  
  ...  
  JOURS = 3;  
  ...  
}
```



Le compilateur réserve de la place en mémoire (ici 4 octets).



/*ERREUR !*/ On ne peut changer la valeur d'une const.

Identificateurs

Les identificateurs nomment les objets C (fonctions, variables ...)

C'est une suite de lettres ou de chiffres.

Le premier caractère est obligatoirement une lettre.

Le caractère _ (souligné) est considéré comme une lettre.

Reconnaissance suivant les 31 premiers caractères.

Le C distingue les minuscules des majuscules.



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)

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

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

Les opérateurs arithmétiques

- Le C propose les opérateurs suivants :

+	addition
-	soustraction
*	multiplication
/	division
%	modulo (reste de la division entière)

% ne peut être utilisé qu'avec des entiers

$7/2$  3

$7.0/2$
 $7/2.0$
 $7.0/2.0$   3.5

Utiliser des opérateurs arithmétiques

Le compilateur considère le type des opérandes pour savoir comment effectuer les opérations

The diagram illustrates how the compiler determines the type of division based on the operand types in the following C code snippet:

```
void main(void)  
{  
    int    i = 5, j = 4, k;  
    double f = 5.0, g = 4.0, h;  
    k = i / j;  
    h = f / g;  
    h = i / j;  
}
```

Annotations and arrows explain the operations:

- k = 5/4 = 1**: Points to the assignment `k = i / j;`. Since both `i` and `j` are `int`, integer division is performed, resulting in 1.
- h = 5.0/4.0 = 1.25**: Points to the assignment `h = f / g;`. Since both `f` and `g` are `double`, floating-point division is performed, resulting in 1.25.
- h = 5/4 = 1.0000**: Points to the assignment `h = i / j;`. Since `i` and `j` are `int`, integer division is performed first (5/4 = 1), and then the result is converted to `double` (1.0000).

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 exclusif)
!	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)
```

```
c >= 10*(i + f)      =====> 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 = (1) si expr1=(1) ou expr2=(1) et faux sinon.

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

Contractions d'opérateurs

- Il y a une famille d'opérateurs

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

- Pour chacun d'entre eux

$\text{expression1 } op= \text{expression2}$

est équivalent à:

$(\text{expression1}) = (\text{expression1}) op (\text{expression2})$

a += 32;

a = a + 32;

f /= 9.2;

f = f / 9.2;

i *= j + 5;

i = i * (j + 5);

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;
```

Préfixe et Postfixe

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

```
int main(void)
```

```
{
```

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

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

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

```
    j = 5;
```

```
    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=6, j=6

i=5, j=6

Les Conversions de types

Le langage C permet d'effectuer des opérations de conversion de type. On utilise pour cela l'opérateur de "cast" ().

```
#include <stdio.h>
#include <conio.h>
void main()
```

```
{
```

```
    int i=0x1234, j;
```

```
    char d,e;
```

```
    float r=89.67,s;
```

```
    j = (int)r;
```

```
    s = (float)i;
```

```
    d = (char)i;
```

```
    e = (char)r;
```

```
    printf("Conversion float -> int: %5.2f -> %d\n",r,j);
```

```
    printf("Conversion int -> float: %d -> %5.2f\n",i,s);
```

```
    printf("Conversion int -> char: %x -> %x\n",i,d);
```

```
    printf("Conversion float -> char: %5.2f -> %d\n",r,e);
```

```
    printf("Pour sortir frapper une touche ");
```

```
    getch();    // pas getchar
```

```
}
```

Conversion float -> int: 89.67 -> 89

Conversion int -> float: 4660 -> 4660.00

Conversion int -> char: 1234 -> 34

Conversion float -> char: 89.67 -> 89

Pour sortir frapper une touche

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 structures de contrôle en C

Les décisions - if then else

Pas de then en C

Le bloc " else " est optionnel.

```
if (expression booléenne vraie)
{
    BLOC 1 D'INSTRUCTIONS
}
else
{
    BLOC 2 D'INSTRUCTIONS
}
```

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

* Tout ce qui est **0** (`'0'` `0` `0.0000` `NULL`) est **faux**

* Tout ce qui est **!= de 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");
```

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 reçu 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 ;  
}
```

if(delta != 0) = if(delta)

if(delta == 0) = if(!delta)

!



Attention!

- Ne pas confondre == (opérateur logique d'égalité) et = (opérateur d'affectation)

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

```
int    main(void)  
{
```

```
    int    i = 0;
```

```
    if(i = 0)    /* ici affectation */  
        printf("i = zero\n");
```

```
    else
```

```
        printf("Quand i != de zero\n");
```

```
    return 0;
```

```
}
```



Quand i != de zero

if emboîtés

- 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");
```

i is reasonable

```
int i = 100;
```

```
if(i > 0) {
```

```
    if(i > 1000)
```

```
        printf(" i > 1000 \n");
```

```
} else
```

```
    printf("i is negative\n");
```

Les itérations – for

```
for( init ; test; increment)
{
    /* corps de for */
}
```

```
int i,j;

for (i = 0; i < 3; i++) {
    printf ( "i = %d\n", i);
}

for(j = 5; j > 0; j- -)
    printf("j = %d\n", j);
```

i = 0
i = 1
i = 2
j = 5
j = 4
j = 3
j = 2
j = 1

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

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

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

(Boucles)

LA BOUCLE TANT QUE ... FAIRE ...

Boucle pré-testée

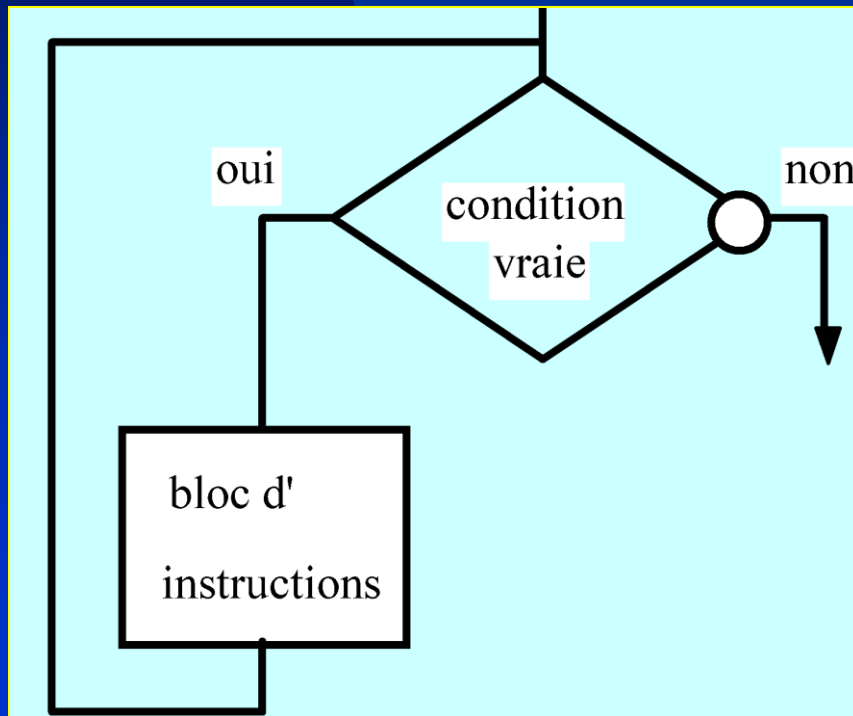


Il s'agit de l'instruction **while** :

tant que (expression vraie)
faire{BLOC D'INSTRUCTIONS}

tant que, pas jusqu'à ce que!

Organigramme:



Syntaxe en C:



while (expression)

```
{  
.....; /* bloc d'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

```
i=1;
while(i<5)
{
    printf("Intérieur %d\n",i);
    i++;
}
printf("Extérieur %d\n",i);
```

Intérieur	1
Intérieur	2
Intérieur	3
Intérieur	4
Extérieur	5

itération

tant que, pas jusqu'à ce que!

```
int j = 5;
```

```
printf("start\n");
while(j == 0)
    printf("j = %d\n", j--);
printf("end\n");
```

start
end

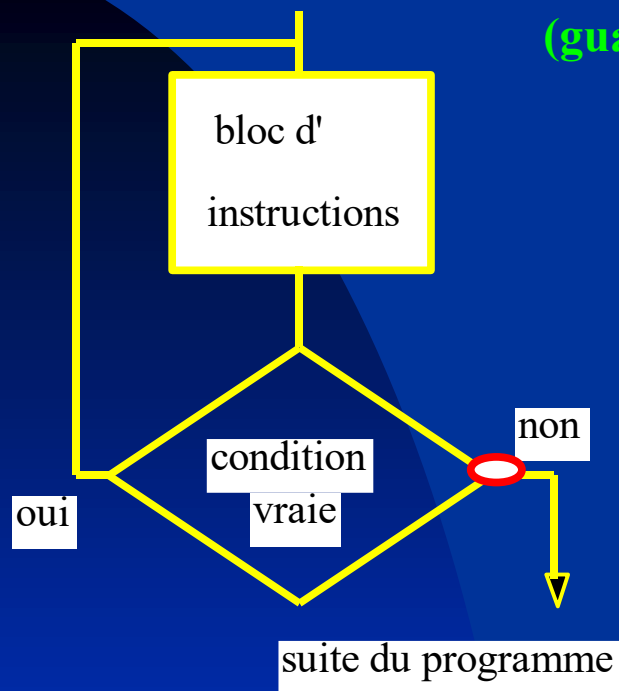
```
i=1;
while(i<5);
{
    printf("Intérieur %d\n",i);
    i++;
}
```

"tant que l'expression est vraie
attendre".

(Boucles)

do while = REPETER ... tant que

(garantit l'exécution au moins une fois)



```
do
{
.....; /* bloc d'instructions */
.....;
}
while (expression);
```

```
int j = 5;
do
    printf("j = %i\n", j--);
while(j > 0);
printf("stop\n");
```

j = 5
j = 4
j = 3
j = 2
j = 1
stop

```
i=1;
do
{
    printf("i=%d ",i);
    i++;
}while(i<0);
printf("stop\n");
```

i = 1
stop

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":

```
char choix;  
printf("SAISIE TAPER 1\n");  
printf("AFFICHAGE TAPER 2\n");  
printf("POUR SORTIR TAPER S\n");  
printf("\nVOTRE CHOIX: ");  
choix = getchar();  
switch(choix)  
{  
    case '1': .....;  
        break;  
  
    case '2': .....;  
        break;  
  
    case 'S': printf("\nFIN DU PROGRAMME ....");  
        break;  
  
    default; printf("\nCE CHOIX N'EST PAS PREVU "); /* pas de break ici */  
}
```

```
int choix;  
  
scanf(" %d ", &choix);  
switch(choix)  
{  
    case 1: ...  
    ...  
}
```



```
float f;  
  
switch(f) {  
    case 2:  
        ...  
}
```

```
switch(i) {  
    case 2 * j:  
        ....  
}
```

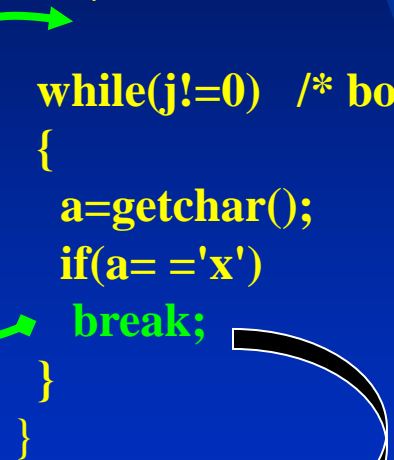
Instructions d'échappement

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);

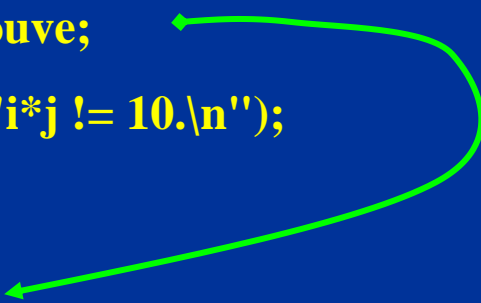
permet de sortir de la fonction qui la contient

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

goto étiquette

```
#include <stdio.h>

void main()
{
    int i, j;
    for (i=0; i < 10; i++)
        for (j=0; j < 4; j++) {
            if ( (i*j) == 10)
                goto trouve;
            printf("i*j != 10.\n");
        }
    trouve:
    printf("i*j = %d * %d = %d == 10.\n", i, j, i*j);
}
```



Tableaux : Introduction

Déclaration de tableaux

- Un tableau (array) est une collection de variables de même type, appelées éléments
- On les déclare par un type, un nom et une dimension (CONSTANTE) placée entre []
- Le C alloue toujours un tableau dans une zone contigüe de la mémoire
- Une fois déclaré, on ne peut redimensionner un tableau

Exemples

int tab[4]; déclare un tableau de 4 valeurs entières

tab[0]	tab[1]	tab[2]	tab[3]
--------	--------	--------	--------



```
#define      SIZE      10  
int         a[SIZE];    /* a est un vecteur de 10 entiers */
```

le compilateur réserve **SIZE** places en mémoire pour ranger les éléments du tableau.

```
float       A[5]   = { 10.1, 20.3, 30.5, 40.0, 50.4 };
```

Accès aux éléments d'un tableau

- Les éléments sont numérotés de 0 à dim-1
- Il n'y a pas de vérification de bornes



```
void main(void)
```

```
{
```

```
    int    a[6];
```

```
    int    i = 7;
```

```
    a[0] = 59;
```

```
    a[5] = -10;
```

```
    a[i/2] = 2;
```

```
    a[6] = 0;
```

```
    a[-1] = 5;
```

```
}
```

a

59

?

?

2

?

-10

0

1

2

3

4

5

```
void main(void)
```

```
{
```

```
    int i;
```

```
    int A[6] = { 1, 2,3, 5, 7, 11 };
```

```
    for (i=0;i<6;i++)
```

```
        printf("%d ", A[i]);
```

```
}
```

Les tableaux consomment beaucoup de place mémoire.
On a donc intérêt à les dimensionner au plus juste.

```
void main()
{
    const int N=10;
    int t[N],i;
    for (i=0;i<N;i++){
        printf("Entrez t[%d]=",i);
        scanf("%d",&t[i]);
    }
}
```

Fonctions en C

Encapsule un traitement particulier formant un tout

- Peut implémenter la notion de module en logique
- Notion de librairie
- Augmente la lisibilité d'un programme
- Réalise un objectif précis
- Améliore le débogage et la maintenance d'un programme

Son utilisation se décompose en trois phases :

- Définition de la fonction
- Déclaration de la fonction
- Appel de la fonction

Déclarer une fonction

Appeler une fonction

Règles de visibilité des variables

Passage des paramètres par valeur

Fonction renvoyant une valeur au programme

Passage des paramètres par valeur et par adresse

Passage des tableaux aux fonctions

Fonctions en C

Les principes

- En C, tout est **fonction** (Définition, Déclaration et Appel).
- Une fonction peut avoir autant de paramètres que l'on veut, même aucun
(comme `void main(void)`)
- Une fonction renvoie une valeur ou aucune.
- Les variables déclarées dans une fonction sont locales à cette fonction
- Une fonction possède un et un seul point d'entrée, mais éventuellement plusieurs points de sortie (à l'aide du mot **return**).
- L'imbrication de fonctions n'est pas autorisée:
une fonction ne peut pas être déclarée à l'intérieur d'une autre fonction. Par contre, une fonction peut **appeler** une autre fonction. Cette dernière doit être déclarée **avant** celle qui l'appelle.

Déclarer une fonction

TYPE de la valeur de retour

3 doubles comme paramètres

`int print_table(double start, double end, double step)`

`{`

`double d;
int lines = 1;`

`printf("Celsius\tFahrenheit\n");
for(d = start; d <= end; d += step, lines++)
 printf("%.1lf\t%.1lf\n", d, d * 1.8 + 32);`

`return lines;`

`}`

Nom de la fonction

Valeur renvoyée

Appeler une fonction

IMPORTANT: cette instruction spécifie comment la fonction est définie

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

```
int print_table(double, double, double);
```

```
void main(void)
```

```
{
```

```
    int    combien;
```

```
    double end = 100.0;
```

```
    combien = print_table(1.0, end, 3);
```

```
    print_table(end, 200, 15);
```

```
}
```

Le compilateur attend des doubles; les conversions sont automatiques

Ici, on ne tient pas compte de la valeur de retour


```

float calculeFloat(float, float);
int addition();

void main(void)
{
    int Val ;

    Val = addition();
    printf("val = %d", Val);
}

int addition()
{
    float tmp;
    tmp = calcule(2.5,3) + calcule(5,7.2);
    return (int)tmp;
}

float calcule(float float1, float float2)
{
    return ( (float1 + float2 ) / 2) ;
}

```

Règles de visibilité des variables

Le C est un langage structuré en blocs { }, les variables ne peuvent être utilisées que là où elles sont déclarées

```
void func(int x);  
int glo=0;           // variable globale  
void main(void)  
{  
    int i = 5, j, k = 2; //locales à main  
    float f = 2.8, g;  
  
    d = 3.7;  
    func (i);  
}  
  
void func(int v)  
{  
    double d, e = 0.0, f=v; //locales à func  
  
    i++; g--; glo++;  
    f = 0.0;  
}
```



Le compilateur ne connaît pas **d**



Le compilateur ne connaît pas **i** et **g**



C'est le **f** local qui est utilisé

Passage des paramètres par valeur

- * Quand une fonction est appelée, ses paramètres sont copiés (passage par valeurs)
- * La fonction travaille donc sur des copies des paramètres et ne peut donc les modifier
- * En C, le passage des paramètres par référence se fait en utilisant des pointeurs (voir plus loin)
- * `scanf()` travaille avec pointeur. C'est le pourquoi du `&`

Passage des paramètres par valeur

```
#include <stdio.h>

void change(int v);

void main(void)
{
    int var = 5;

    change(var);

    printf("main: var = %d\n", var);
}

void change(int v)
{
    v *= 100;
    printf("change: v = %d\n", v);
}
```

change: v = 500
main: var = 5

FONCTION RENVOYANT UNE VALEUR AU PROGRAMME

```
#include <stdio.h>

int  change(int v);

void main(void){
    int var = 5;
    int valeur;
    valeur = change(var);

    printf('main: var = %d\n', var);
    printf('main: valeur = %d\n', valeur);
}

int change(int v)
{
    v *= 100;
    printf('change: v = %d\n', v);
    return (v+1);
}
```

change: v =
main: var =
main: valeur =

Une fonction se termine et « rend la main » au code appelant lorsque son exécution rencontre l'instruction : `return expression;` ou `return;`

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

```
int return_Val(int v);
```

```
void main(void){  
    int var = 5;  
    int valeur;  
    valeur = return_Val(var);  
  
    printf("main: var = %d\n", var);  
    printf("main: valeur = %d\n", valeur);  
}
```

```
int return_Val(int v)  
{  
    if (v == 10)    return (2*v);  
    else            return (3*v);  
}
```

return_Val : rien
main: var =
main: valeur =

```
int max (int a, int b) {  
    if (a > b)    return a ;  
    else return b ;  
}  
  
int min (int a, int b) {  
    if (a < b)    return a ;  
    else return b ;  
}  
  
int difference (int a, int b, int c) {  
    return (max (a, max(b,c)) - min (a, min (b, c))) ;  
}  
  
void main ()  
{  
    printf ("%d", difference (10, 4, 7)) ;  
}
```

Une fonction est dite **réursive**, dès lors qu'elle se fait appel pour calculer une sous partie de la valeur finale qu'elle retournera.

```
#include <stdio.h>
int fact(int n);
int return_Val(int v);

void main(void){
    int var = 5, int valeur;
    valeur = return_Val(var);
    printf("main: var = %d\n", var);
    printf("main: valeur = %d\n", valeur);
}

int fact(int n) {
    return (n<2?1:((int)n*fact(n-1)));
}
int return_Val(int v)
{
    if (v == 10)    return (2*v);
    else            return fact(v);
}
```

return_Val : rien
main: var =
main: valeur =

(0)=0*fact(-1)

int temp=1;
while(n>1) temp*=n--;

L'opérateur ternaire “?:”

```
return(n<2?1:((int)n*fact(n-1)));
```

Cet opérateur s'écrit: (condition) ? (instruction_1) : (instruction_2)

si la condition est vérifiée le bloc “instruction_1” est exécuté, sinon c'est le bloc “instruction_2” qui l'est.

```
if (n<2 ) {  
    return 1;  
}  
else {  
    return ((int)n*fact(n-1));  
}
```

```
return(n<2?1:((int)n*fact(n-1)));
```

```
int fact( int n) {  
    if (n < 1) return 1;  
    else return ( n * fact(n-1) );  
}
```

```
void hanoi(int n, int a,  
int b){  
    int c ;  
    ...  
    if (n == 1) move(a,b);  
    else {  
        hanoi(n-1, a, c);  
        move (a, b);  
        hanoi(n-1, c, b);  
    }  
}
```

Passer des tableaux aux fonctions



- Les tableaux peuvent être passés comme paramètres d'une fonction.
- Ils ne peuvent pas être retournés comme résultat d'une fonction.
- La longueur du tableau ne doit pas être définie à la déclaration de la fonction.
- Un tableau peut être modifié dans une fonction. Il est passé par référence (adresse) et non par valeur.

```
#include <stdio.h>
void  somme(int x[], int n);
void  main(void){
    int i;
    int p[6] = { 1, 2,3, 5, 7, 11 };
    somme(p, 6);
    for (i=0;i<6;i++)
        printf("%d ", p[i]);
}

void  somme(int a[], int n){
    int    i;
    for(i = n-1; i >0 ; i--)
        a[i] += a[i-1];
}
```

p avant somme

1
2
3
5
7
11

p après somme

1
3
5
8
12
18

Exemple: transposition d'une matrice

```
void transpose ( float a[][20], int m, int n);
```

```
void main(void)
{ float matrice [100][20]; /* matrice (100,20) */
  /* ... */
  transpose (matrice, 3, 2);
}
```

```
void transpose ( float a[][20], int m, int n)
{ int i,j;
  float tmp;
  for (i=0;i<m;i++)
    for(j=0;j<i;j++) {
      tmp = a[i][j];
      a[i][j]=a[j][i];
      a[j][i]=tmp;
    }
}
```

TP1

Écrire un programme en C qui range au maximum (taille Nmax) 20 nombres entiers saisis au clavier dans un tableau. Il doit gérer en boucle le menu de choix suivant :

- A- Saisie et affichage**
- B- Moyenne**
- C- Suppression du Max et affichage**
- D- Suppression du Min et affichage**
- E- Ajout d'un entier à une position donnée**
- Q- Quitter**

Le point **A** sera traité par deux fonctions :

Saisie : la saisie au clavier des entiers et leur rangement dans le tableau.

Dans cette fonction on demandera le nombre d'éléments (**NE** ≤ **Nmax**) à saisir.

Affichage : affichage des données.

Le point **B** sera traité par une fonction :

Moyenne : fonction de calcul de la moyenne du tableau (avec affichage du résultat).

Le point **C** sera traité par trois fonctions :

Max_elem : une fonction qui retourne la position du maximum des valeurs du tableau.

Supprimer : supprime le Max du tableau.

Affichage : affichage des données.

Le point **D** sera traité par trois fonctions :

Min_elem : une fonction qui retourne la position du minimum des valeurs du tableau.

Supprimer : supprime le Mix du tableau.

Affichage : affichage des données.

Le point **E** sera traité par deux fonctions :

Ajout : c'est une fonction où on demande à l'utilisateur d'introduire l'entier et la position.

Puis vous insérer l'entier dans le tableau à la position indiquée.

Affichage : affichage des données.

Strings

- Le C n'a pas de type string, on utilise donc des tableaux de char.
- Un caractère spécial, le caractère “null”, marque la fin du string (à ne pas confondre avec le pointeur NULL).
- Ce caractère est noté ‘\0’, de code ASCII 0.
- Ce caractère est soit ajouté automatiquement par le compilateur, soit à introduire par le programmeur, dépendant des fonctions utilisées.

```
char first_name[5] = { 'J','o','h','n','\0'};
```

```
char last_name[6] = "Minor";
```

```
char other[] = "Tony Bloot";
```

```
char characters[7] = "No null";
```

'J'	'o'	'h'	'n'	0
-----	-----	-----	-----	---

'M'	'i'	'n'	'o'	'r'	0
-----	-----	-----	-----	-----	---

'T'	'o'	'n'	'y'	32	'B'	'l'	'o'	'o'	't'	0
-----	-----	-----	-----	----	-----	-----	-----	-----	-----	---



'N'	'o'	32	'n'	'u'	'l'	'l'
-----	-----	----	-----	-----	-----	-----



Pas de ‘\0’

Entrées/Sorties Standards



- **getchar()** :  lire un caractère sur entrée standard (*valeur -1 = EOF = fin*)
- **putchar()** :  écrire un caractère sur sortie standard

Exemple: Copie de Input sur Output.

```
main(){
    char c;
    c = getchar();
    while (c != EOF) {
        putchar(c);
        c = getchar();
    }
}
```

```
main(){
    char c;
    while ( (c=getchar())!= EOF)
        putchar(c);
}
```

- Fichiers stdin et stdout: Tout programme lit ou écrit sur ces fichiers (défaut *clavier* et *écran*).

Affichage à l'écran des strings

- Soit caractère par caractère, avec le format “%c” ou putchar,
- ou, plus simplement, avec le format “%s” ou puts .

```
char last_name[6] = "Minor";
```

```
int i = 0;
```

```
last_name[3] = '\0';
```

```
while(last_name[i] != '\0')
```

```
    printf("%c", last_name[i++]);
```

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

Minor

'M'	'i'	'n'	'o'	'r'	0
-----	-----	-----	-----	-----	---

Min

} printf("%s\n", last_name);
puts(last_name);

Saisie des strings

```
char texte[30];  
printf("ENTRER UN TEXTE: ");  
scanf("%s", texte);           //est équivalent à  gets(texte);
```



Une chaîne : pas de symbole **&**. On utilisera de préférence la fonction **gets** non formatée.



Si text = un deux → ≠ entre scanf et gets?

RQ:

scanf ne permet pas la saisie d'une chaîne comportant des espaces: les caractères saisis à partir de l'espace ne sont pas pris en compte. Pour saisir une chaîne de type "il fait beau", il faut utiliser **gets**.

Affectation de strings :

Les strings peuvent être initialisés avec "=", mais pas affectés avec "=" (ce sont des tableaux!)

```
#include <stdio.h>

#include <string.h>

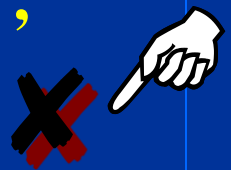
void main(void) {

    char who[] = "Tony Bloot";

    who = "John Minor";

    strcpy(who, "John Minor");

}
```



Fonctions permettant la manipulation des chaînes

string.h ou stdlib.h

void *strcat(char *chaine1, char *chaine2)

//Concatène les 2 chaînes, résultat dans chaine1.

void *strcpy(char *chaine1, char *chaine2)

//Copie la chaine2 dans chaine1. + **'\0'**

void *strncpy(char *chaine1, char *chaine2, NCmax)

// **idem strcpy mais** limite la copie au nombre de caractères NCmax.

int strcmp(char *chaine1, char *chaine2)

//Compare les chaînes de caractères chaine1 et chaine2,
renvoie un nombre:

- positif si la chaîne1 est supérieure à la chaîne2 (au sens de l'ordre alphabétique)
- négatif si la chaîne1 est inférieure à la chaîne2
- nul si les chaînes sont identiques.

int strlen(char *chaine)

// renvoie la longueur de la chaine (''\0' non comptabilisé).

int atoi(const char *s)

// convertit l'argument s en un **int**,

Exemple:

```
#include <string.h>
#include <ctype.h>           // toupper
#define EOL  '\n'

void main(void)
{
    int L,i=0;
    char s[9] = "midi", u[11] = "avant", v[10] = "soir";
    char text[80];

    While ( (text[i] = getchar() ) != EOL ) i++;
    puts(text);

    L = strlen(s);           /* L = 4 */
    for (i=0;i<L;i++) {
        putchar(toupper(s[i]));    /* M, I, D, I */
    }

    strcat(u,"-");           /* u devient "avant-" */
    strcat(u , s);           /* u devient "avant-midi" */
    if (strcmp (u, s) >=0 )   /* "avant-midi" < "midi" */
        printf ("il y a une erreur\n");
    else {
        printf("%s\n", u);     /* avant-midi */
        strcpy (u, v);
        printf("%s\n", u);     /* soir */
    }
}
```

```
text[i] = getchar();
while ( text[i] != EOL ) {
    i++;
    text[i] = getchar();
}
```

Passage des paramètres par valeur et par adresse

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

```
void ech(int x,int y){  
    int tampon;  
    tampon = x;  
    x = y;  
    y = tampon;  
}
```

```
void main() {  
    int a = 5 , b = 8;  
    ech(a,b);  
    printf(" a=%d\n ", a) ;  
    printf(" b=%d\n ", b) ;  
}
```

PASSAGE DES PARAMETRES

PAR VALEUR

a et b: variables locales à main(). La fonction ech ne peut donc pas modifier leur valeur. On le fait donc en passant par l'adresse de ces variables.

a = 5

b = 8

Syntaxe qui conduit à une erreur

🕯 Les Pointeurs en C 🕯

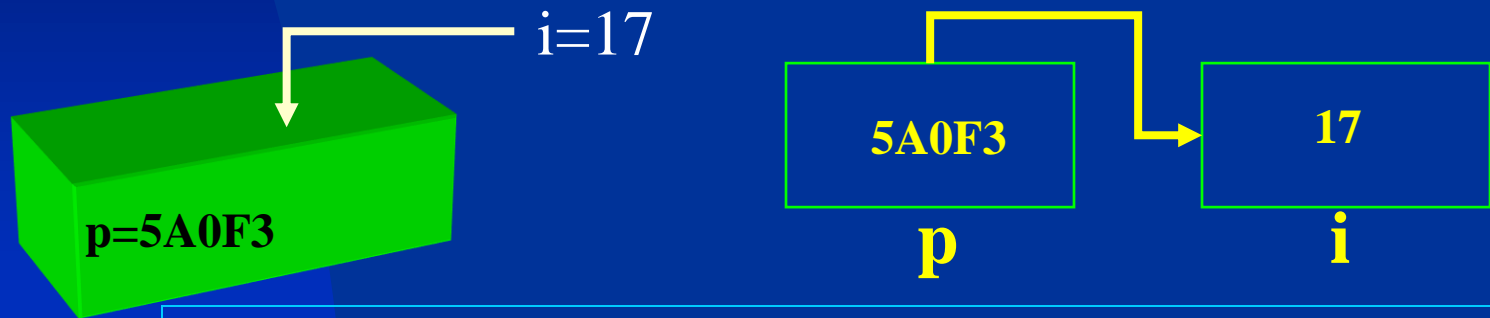
- La déclaration de pointeurs
- Valeurs pointées et adresses
- Passage de paramètres de fonction par référence
- Pointeurs et tableaux

Les pointeurs, c'est quoi?

Un pointeur est une variable particulière, dont la valeur est l'adresse d'une autre variable.

- Pointeur p: valeur 5A0F3 (adresse hexadécimale)
- Adresse 5A0F3: valeur 17 (correspondant à la valeur d'un entier i)

En accédant à cette adresse, on peut accéder indirectement à la variable et donc la modifier.



Un pointeur est une adresse mémoire. On dit que le pointeur p pointe vers i, puisque p pointe vers l'emplacement mémoire où est enregistrée i.

Les pointeurs: pourquoi ?

- Les pointeurs sont nécessaires pour:
 - ◆ effectuer les appels par référence (i.e. écrire des fonctions qui modifient certains de leurs paramètres)
 - ◆ manipuler des structures de données dynamiques (liste, pile, arbre,...)
 - ◆ allouer dynamiquement de la place mémoire

Déclaration de Pointeurs

Le symbole `*` est utilisé entre le type et le nom du pointeur

- Déclaration d'un entier:

```
int i;
```

- Déclaration d'un pointeur vers un entier:

```
int *p;
```

Exemples de déclarations de pointeurs

```
int *pi;          /* pi est un pointeur vers un int
                  *pi désigne le contenu de l'adresse */
float *pf;        /* pf est un pointeur vers un float */

char c, d, *pc;   /* c et d sont des char*/
                  /* pc est un pointeur vers un char */
double *pd, e, f; /* pd est un pointeur vers un double*/
                  /* e et f sont des doubles */
double **tab;     /* tab est un pointeur pointant sur un pointeur qui
                  pointe sur un flottant double */
```

Opérateurs unaires pour manipuler les pointeurs, & (adresse de) et * (contenu)

Exemple: `int i = 8;`

`printf("VOICI i: %d\n",i);`

`printf("VOICI SON ADRESSE EN HEXADECIMAL: %p\n",&i);`

```
void main(void)
```

```
{
```

```
    char c = 'a', d = 'z';
```

```
    char *p;
```

```
    p = &c;
```

```
    printf("%c\n", *p);
```

```
    p = &d;
```

```
    printf("%c\n", *p);
```

```
}
```

`nom_de_Pointeur = &nom_de_variable`

reçoit l'adresse de c; donc pointe sur c

`*p = c;`



a
z

L'opérateur * ("valeur pointée par")

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

```
void main() {
```

```
    int *p, x, y;
```

```
    p = &x;                                /* p pointe sur x */
```

```
    x = 10;                                /* x vaut 10 */
```

```
    y = *p - 1;    printf(" y= *p - 1 =?  = %d\n" , y);
```

y vaut ?

```
    *p += 1;    printf(" *p += 1 =? *p = x= ?  = %d %d\n" , *p, x);
```

x vaut ?

```
    (*p)++;    printf(" (*p)++ =? *p = x= ?  = %d %d  alors y=%d \n" , *p, x, y);
```

incrémente aussi de 1 la variable pointée par p, donc x vaut ??.
y vaut 9

```
    *p=0;    printf(" *p=0  x=? = %d\n" , x);
```

comme p pointe sur x, maintenant x vaut ?

```
    *p++; *p=20; printf(" *p++  x=? = %d\n" , x);
```

comme p ne pointe plus sur x, x vaut tjr ?

```
}
```

Utiliser des pointeurs

- On peut donc accéder aux éléments par pointeurs en faisant des calculs d'adresses (addition ou soustraction)

```
long  v[6] = { 1,2,  
              3,4,5,6 };
```

```
long  *p;
```

```
p = v;
```

```
printf("%ld\n", *p);
```

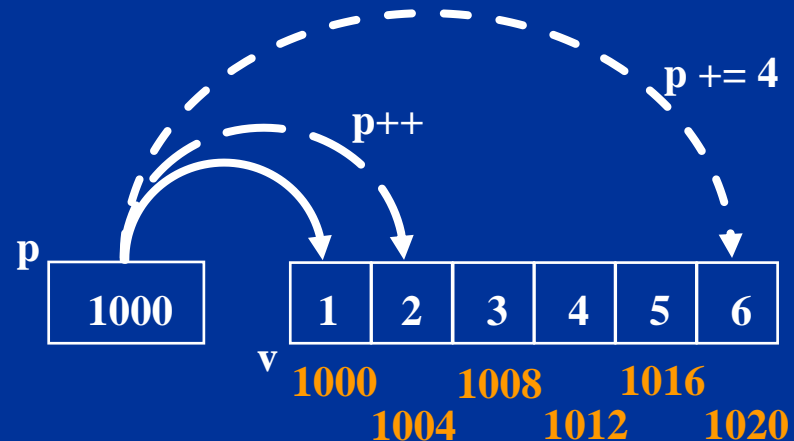
```
p++;
```

```
printf("%ld\n", *p);
```

```
p += 4;
```

```
printf("%ld\n", *p);
```

1
2
6



* et ++

*p++ signifie:

*p++ trouver la valeur pointée

*p++ passer à l'adresse suivante

(*p)++ signifie:

(*p)++ trouver la valeur pointée

(*p)++ incrémenter cette valeur (sans
changer le pointeur)

*++p signifie:

*++p incrémenter d'abord le pointeur

*++p trouver la valeur pointée



Passage des paramètres par valeur et par adresse

Syntaxe qui conduit à une erreur:

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

```
void ech(int x,int y)
{
    int tampon;
    tampon = x;
    x = y;
    y = tampon;
}
```

```
void main()
{
    int a = 5 , b = 8;
    ech(a,b);
    printf(" a=%d\n ", a);
    printf(" b=%d\n ", b);
}
```

PASSAGE DES PARAMETRES
PAR VALEUR

a et b:
variables
locales à
main(). La
fonction ech
ne peut donc
pas modifier
leur valeur.
On le fait
donc en
passant par
l'adresse de
ces variables.

a = ?
b = ?

Syntaxe correcte:

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

```
void ech(int *x,int *y)
{
    int tampon;
    tampon = *x;
    *x = *y;
    *y = tampon;
}
```

```
void main()
{
    int a = 5 , b = 8 ;
    ech(&a,&b);
    printf(" a=%d\n ", a);
    printf(" b=%d\n ", b);
}
```

PASSAGE DES PARAMETRES
PAR ADRESSE

a = ?
b = ?

Passage de paramètres de fonction par référence ou adresse

- Quand on veut modifier la valeur d'un paramètre dans une fonction, il faut passer ce paramètre par référence ou adresse
- En C, cela se fait par pointeur

```
void change ( int *a, int *b)
{
    int tmp;
    tmp = *a;
    *a = *b;
    *b = tmp;
}
```


Appel des fonctions par référence

```
#include <stdio.h>
void change(int* p);
void main(void)
{
    int var = 5;
    change(&var);
    printf("main: var = %d\n",var);
}

void change(int* p)
{
    *p *= 100;
    printf("change: *p = %d\n",*p);
}
```



change: *p = 500
main: var = 500

```

#include <stdio.h>

void somme(int , int , int *);
int modif(int , int *, int *);
void main(){
    int a, b, c;
    a = 2;  b = 8;
    somme(a, b, &c);
    printf("Somme de a=%d et b=%d : %d\n",a, b, c);
    a = modif(a, &b, &c);
    printf(" Modif : a=%d , b=%d et c= %d\n",a, b, c);
}

void somme(int x, int y, int *z){
    *z = x + y;
}

int modif(int x, int *y, int *z){
    x *= 2;    *y= x+ *y; *z= 5;
    return x;
}

```

Somme de a=? et b=? : ?

Modif : a=?, b=? et c= ?

Identification des tableaux et pointeurs

- En C, le nom d'un tableau représente l'adresse de sa composante 0, donc `a == &a[0]`
- C'est pour cela que les tableaux passés comme paramètres dans une fonction sont **modifiables**

Passer des tableaux aux fonctions

- Pour le compilateur, un tableau comme argument de fonction, c'est un pointeur vers sa composante 0 (à la réservation mémoire près).
- La fonction peut donc modifier n'importe quel élément (passage par référence)
- Le paramètre peut soit être déclaré comme tableau, soit comme pointeur

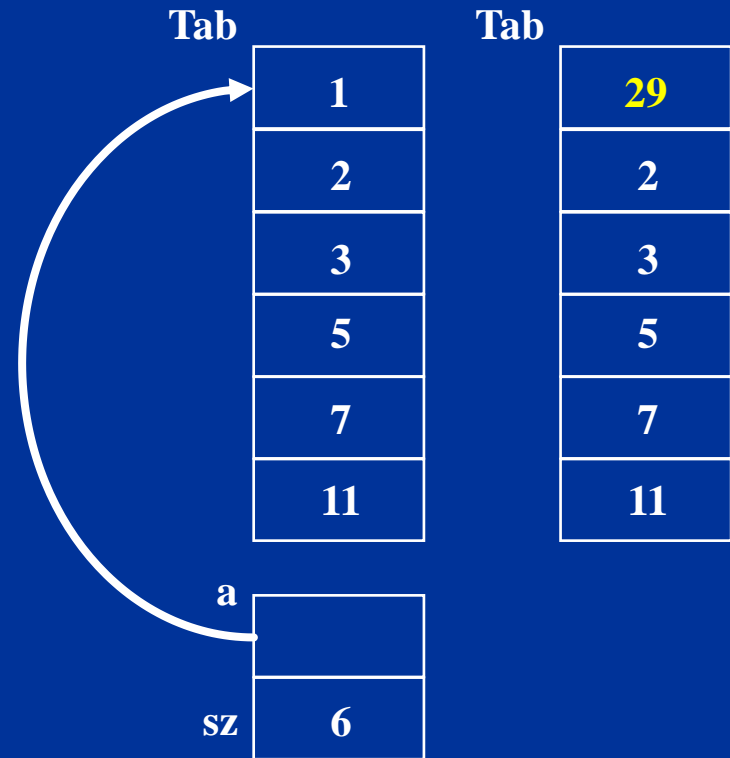
```
int add_elements(int a[], int size)
{
```

```
int add_elements(int *p, int size)
{
```

Example

```
#include <stdio.h>
void sum(long [], int);
int main(void)
{
    long Tab[6] = { 1, 2,
                    3, 5, 7, 11 };
    sum(Tab, 6);
    printf("%ld\n", Tab[0]);
    return 0;
}

void sum(long a[], int sz)
{
    int i;
    long total = 0;
    for(i = 0; i < sz; i++)
        total += a[i];
    a[0] = total;
}
```



Utiliser des pointeurs - exemple

```
#include <stdio.h>

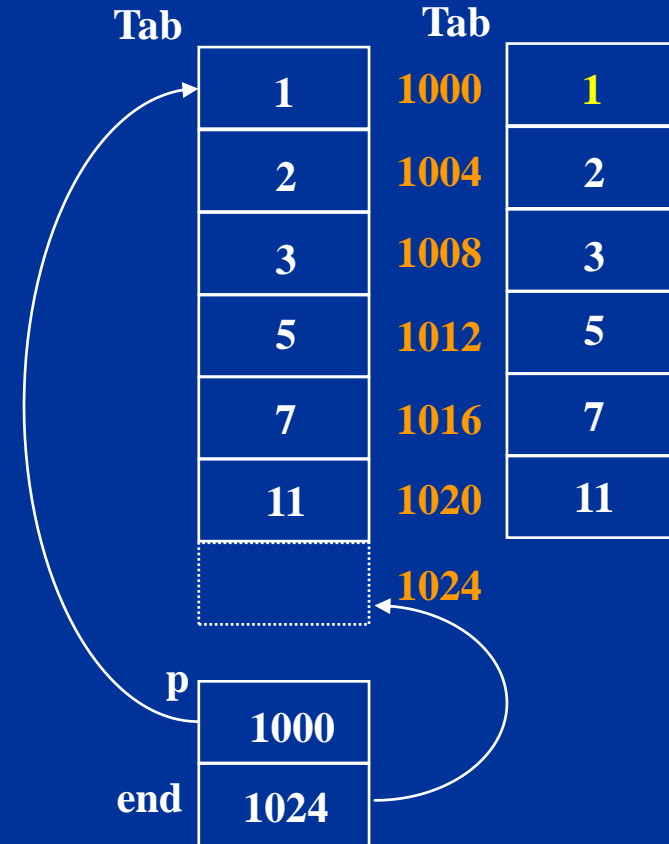
long  sum(long*, int);

int  main(void)
{
    long  Tab[6] = { 1, 2,
                    3, 5, 7, 11 };
    printf("%ld\n", sum(Tab, 6));
    return 0;
}

long  sum(long *p, int sz)
{
    long  *end = p + sz;
    long  total = 0;

    while(p < end)
        total += *p++;

    return total;
}
```



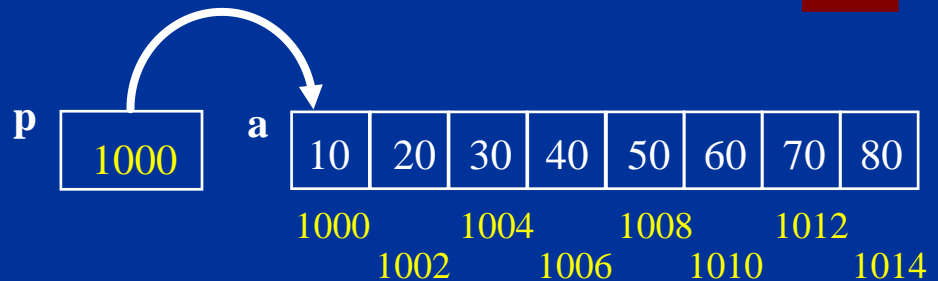
Quelle notation?

- $A[0]$ est équivalent à $*A$
- $A[i]$ est équivalent à $*(A + i)$
- $\&A[0]$ est équivalent à A

```
short  a[8] = { 10, 20, 30, 40, 50, 60, 70, 80 };  
short  *p = a;
```

```
printf("%d\n", a[3]);  
printf("%d\n", *(a + 3));  
printf("%d\n", *(p + 3));  
printf("%d\n", p[3]);
```

40
40
40
40



SCANF

Pour saisir un caractère ou un nombre

```
char c;  
float Tab[10], X[5][7]; // *Tab,(*X)[7]  
printf("TAPER UNE LETTRE: ");  
scanf("%c",&c);  
scanf("%f",&Tab[5]);
```

On saisit ici le contenu de l'adresse &c c'est-à-dire le caractère c lui-même.

pointeur

```
char *adr;  
printf("TAPER UNE LETTRE: ");  
scanf("%c",adr);
```

On saisit ici le contenu de l'adresse adr.

`float x[dim];` réserve un espace de stockage pour dim éléments
`x[0]` stocke l'adresse du premier élément.

`float *x;` ne réserve que le seul espace destiné au pointeur x,
AUCUN espace n'est réservé pour une ou plusieurs variables.

La réservation d'espace est à la charge du programmeur qui doit le faire de façon *explicite*, en utilisant les fonctions standard **malloc()**, ... **qui sont** prototypées dans `<stdlib.h>` et `<alloc.h>`

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

```
void main()
{
```

```
    char *texte;
    float *adr1,*adr2;
```

```
    adr1 = (float*)malloc(4*sizeof(float));
    adr2 = (float*)malloc(10*sizeof(float));
    texte = (char*)malloc(10);
```

```
    *adr1 = -37.28;
    *adr2 = 123.67;
    printf("adr1 = %p adr2 = %p r1 = %f
           r2 = %f\n",adr1,adr2,*adr1,*adr2);
    free(adr1); // Libération de la mémoire
    free(adr2);
    free(texte);
```

```
}
```


sizeof

```
void main()
{
    int i;
    char c;
    float f;
    double d;

    printf ("caractère : %d \n ", sizeof c);
    printf ("entier : %d \n ", sizeof i);
    printf ("réel : %d \n ", sizeof f);
    printf ("double : %d \n ", sizeof d);
}
```

caractère : 1
entier : 2 ou 4
réel : 4
double : 8

EXP : Création d'un tableau de taille quelconque

```
#include <stdio.h>           // pour la fonction printf()
#include <stdlib.h>          // pour la fonction malloc()

void main(){
    int i , dim;              // compteur et taille du tableau
    long* tableau;           // pointeur pour stocker l'adresse du tableau

    scanf ("%d", &dim);      // saisie par l'utilisateur de la taille du tableau
    tableau = (long*) malloc(dim * sizeof(long)); //allocation (dynamique) du tableau

    // remplissage du tableau, on utilise les indices 0 à (dim -1)
    for (i = 0; i < dim; i++)
        tableau[i] = i;

    // affichage du contenu du tableau
    for (i = 0; i < dim; i++)
        printf("%ld\n", tableau[i]);

    // destruction du tableau : libération de la mémoire réservée
    free(tableau);
}
```

La mémoire dynamique varie en cours d'exécution, et peut n'être pas suffisante pour allouer les variables, provoquant le plantage du programme. Il faut donc tester le retour des fonctions d'allocation et traiter les erreurs.

Exemple:

```
#define BUFSIZE 100
long *var;
if ( !(var = (long *) malloc(BUFSIZE * sizeof(long))) )
    // Si échec de réservation de BUFSIZE emplacement de type long ( ) ≠ 0
    {
        fprintf(stderr, "ERREUR espace mémoire insuffisant !\n");
        exit (1); // fin anticipée du programme ; code de retour 1
    }
else {
    // le programme continue
}
```

L'espace alloué doit être libéré par free(), dont l'argument est un pointeur réservé dynamiquement. free(var);

char * malloc(unsigned taille);

réserve taille octets, sans initialisation de l'espace. Retourne un pointeur sur une zone de taille octets.

char * calloc(unsigned nombre, unsigned taille);

réserve nombre éléments de taille octets chacun ; l'espace est initialisé à 0.

```
#define alloue(nb,type)  (type *)calloc(nb,sizeof(type))
```

```
char *s;
```

```
s = alloue(250,char);    // réserve 250 octets initialisés à '\0'
```

void * realloc(void *block, unsigned taille);

modifie la taille affectée au bloc de mémoire fourni par un précédent appel à malloc() ou calloc().

void free(void *block);

libère le bloc mémoire pointé par un précédent appel à malloc(), calloc() ou realloc().

Valable pour réserver de l'espace pour les autres types de données int, float, ...
Elles retournent le pointeur **NULL** (0) si l'espace disponible est insuffisant.

Structures en C

- Concepts
- Créer un type de structure
- Créer une instance de structure
- Initialiser une instance
- Accéder aux membres d'une instance
- Passer les structures comme paramètres
- Listes chaînées

Concepts

- Une structure est une collection de plusieurs variables (champs) groupées ensemble pour un traitement commode
- Les variables d'une structure sont appelées *membres* et peuvent être de n'importe quel type, par exemple des tableaux, des pointeurs ou d'autres structures

```
struct Membre
{
    char    nom[80];
    char    adresse[200];
    int     *numero;
    float    amende[10];
};
```

Les étapes sont:

- ◆ déclarer le type de la structure
- ◆ utiliser ce type pour créer autant d'instances que désirées
- ◆ Accéder les membres des instances

Déclarer les structures

- Les structures sont définies en utilisant le mot-clé struct



```
struct Date
{
    int jour;
    int mois;
    int an;
};
```

```
struct Livre
{
    char titre[80];
    char auteur[80];
    float prix;
};
```

```
struct Membre
{
    char nom[80];
    char adresse[200];
    int numero;
    float amende[10];
    struct Date emprunt;
    struct Date creation;
};
```

```
struct Pret
{
    struct Livre b;
    struct Date due;
    struct Membre *who;
};
```

Déclarer des instances

- Une fois la structure définie, les instances peuvent être déclarées
- Par abus de langage, on appellera structure une instance de structure

```
struct Date
{
    int    jour;
    int    mois;
    int    an;
} hier, demain;
```

```
struct Date paques;
struct Date semaine[7];
```


```
struct Date nouvel_an = { 1, 1, 2001 };
```

← **Déclaration
avant ‘;’ .**


← **Initialisation .**

Des structures dans des structures

```
struct Date
{
    int    jour;
    int    mois;
    int    an;
};
```



```
struct Membre
{
    char    nom[80];
    char    adresse[200];
    int     numero;
    float   amende[10];
    struct  Date emprunt;
    struct  Date creation;
};
```



```
struct Membre    m = {
    "Arthur Dupont",
    "rue de Houdain, 9, 7000 Mons",
    42,
    { 0.0 },
    { 0, 0, 0 },
    { 5, 2, 2001 }
};
```

Accéder aux membres d'une structure

- Les membres sont accédés par le nom de l'instance, suivi de `.`, suivi du nom du membre

```
struct Membre m;
```

```
printf("nom = %s\n", m.nom);
```



```
printf("numéro de membre = %d\n", m.numero);
```

```
printf("amendes: ");
```

```
for(i = 0; (i < 10) && (m.amende[i] > 0.0); i++)
```

```
    printf("%.2f Euros", m.amende[i]);
```



```
printf("\nDate d'emprunt %d/%d/%d\n", m.emprunt.jour,  
        m.emprunt.mois, m.emprunt.an);
```

Différence entre tableaux et structures

	Arrays	Structures
Nom	pointeur élément 0	La structure
passage aux fonctions	pointeur	valeur /pointeur
retour de fonctions	interdit	valeur /pointeur
assignation	interdit	possible

Assignation des structures

- L'opération d'affectation = peut se faire avec des structures
- Tous les membres de la structure sont copiés (aussi les tableaux et les sous-structures)

```
struct Membre  m = {  
    "Arthur Dupont",  
    ....  
};  
  
struct Membre temp;  
  
temp = m;
```

Passer des structures comme paramètres de fonction

- Une structure peut être passée, comme une autre variable, par valeur ou par adresse
- Passer par valeur n'est pas toujours efficace (recopiage à l'entrée)
- Passer par adresse ne nécessite pas de recopiage

```
void  Par_valeur(struct Membre m);  
void  Par_reference(struct Membre *m);  
  
Par_valeur(m);  
Par_reference(&m);
```

Quand la structure est un pointeur !

Utiliser **p->name**



- L'écriture **p->name** est synonyme de **(*p)->name**, où **p** est un pointeur vers une structure

```
void  affiche_membre (struct Membre *p)
{
    printf("nom = %s\n", p->nom);
    printf("adresse = %s\n", p->adresse);
    printf("numéro de membre = %d\n", p->numero);

    printf("\nDate d'affiliation %d/%d/%d\n",
           p->creation.jour, p->creation.mois, p->creation.an);
}
```

Retour de structures dans une fonction

- Par valeur (recopiage) ou par référence

```
struct Complex add(struct Complex a, struct Complex b)
{
    struct Complex result = a;
    result.real_part += b.real_part;
    result.imag_part += b.imag_part;
    return result;
}
```

```
struct Complex  c1 = { 1.0, 1.1 };
struct Complex  c2 = { 2.0, 2.1 };
struct Complex  c3;

c3 = add(c1, c2); /* c3 = c1 + c2 */
```

TP

Enoncé

Ecrire un programme permettant de :

Constituer un tableau de 20 Enseignants max(**NE_max**). La structure est la suivante :

```
struct {  
    char nom_pren[40];           // nom+pren  
    char nom[20];  
    char pren[20];  
    int NH[NM_max] ;           // NM_max=3 : nbre d'heures pr NM_max matières  
}
```

Le programme doit gérer en boucle le menu de choix suivant:

- 1- Saisie et affichage
- 2- Construction et affichage
- 3- Modifier et affichage
- 4- Tri et affichage
- 5- Quitter

Tri à bulles

```
while(???) {
    for j = 0 to .... {
        if tab[j] > tab[j+1] {
            <on échange tab[j] et tab[j+1]>
        }
    }
}
```

0	1	2	3	4	5	6	j
2	6	4	8	12	13	0	tab[j]

6 2
 6 4 2
 6 4 8 2
 6 4 8 12 2

$\text{tab}[j] < \text{tab}[j+1]$

13	12	8	6	4	2	0
----	----	---	---	---	---	---

6	4	8	12	13	2	0
---	---	---	----	----	---	---

Gestion des fichiers en C

Elle est assurée par la librairie standard `<stdio.h>` via un ensemble de fonctions commençant par “f”

- Avant de manipuler un fichier, il faut lui associer un descripteur (pointeur vers la structure fichier)
- Il y a 3 descripteurs automatiquement ouvert par tout programme C, `stdin`, `stdout` et `stderr`
- `stdin` (standard input) est connecté au clavier. On peut y lire
- `stdout` (standard output) et `stderr` (standard error) sont reliés au moniteur. On peut y écrire.



`FILE *f;`

`/*déclare un descripteur f de fichier*/`

Ouvrir et fermer des fichiers

Les fichiers sont ouverts avec la fonction **fopen**
ils sont fermés avec la fonction **fclose**

```
FILE* fopen(const char* name, const char* mode);  
int fclose (FILE *f);
```

```
#include <stdio.h>  
void main(void)  
{  
    FILE*      in;  
    FILE*      out;  
    FILE*      append;  
    in = fopen("autoexec.bat", "r");  
    out = fopen("autoexec.bak", "w");  
    append = fopen("config.sys", "a");  
    /* ... */  
    fclose(in);  
    fclose(out);  
    fclose(append);  
}
```

Lecture et écriture sur fichier

Fonctions de lecture

```
int    fscanf(FILE* stream, const char* format, ...);  
int    fgetc(FILE* stream);  
char*  fgets(char* buffer, int size, FILE* stream);
```

Fonctions d'écriture

```
int    fprintf(FILE* stream, const char* format, ...);  
int    fputc(int ch, FILE* stream);  
int    fputs(const char* buffer, FILE* stream);
```

`int feof(FILE *f); /*renvoie une valeur non nulle si fin de fichier*/`



Exemple d'écriture (lecture) de fichier

```
#include <stdio.h>
void sauvegarde( char titre[], int n,
                 float x[], int ind [] )
{
    int i=0;
    FILE *f;
    f = fopen("monfichier.dat","w");
    if (f != NULL){
        fprintf(f,"%s\n",titre);
        for (i=0; i < n; i++ ) {
            fprintf(f,"%f %d\n", x[i],ind[i]);
        }
    }
    fclose(f);
}
```

Mon titre

3.0 1

4.5 2

7.3 3

```
#include <stdio.h>
void main(void)
{
    char titre[81];
    float x[10];
    int ind[10], i=0;
    FILE *f;
    f = fopen("monfichier.dat","r");
    if (f!= NULL) {
        fgets(titre,80,f);
        while(!feof(f)) {
            fscanf(f,"%f %d",&x[i],&ind[i]);
            i++;
        }
    }
    fclose(f);
}
```

La constante NULL (définie comme 0 dans stdio.h) réfère à une adresse non définie

Compilation Séparée et édition de Lien

Un *programme C* est un ensemble de *fonctions* dans un ou plusieurs fichiers.

Fichier PP.c

```
main(){  
...  
appelle f()  
...  
}  
f(){  
appelle g()...  
}  
g(){  
...  
}
```

Cas I

Fichier PP.c

```
main(){  
...  
appelle f()  
...  
}  
f(){  
appelle g()...  
}
```

Cas II

Fichier SP.c

```
g(){  
...  
}
```

Structure d'un programme C

```
#include <stdio.h>
#define DEBUT -10
#define FIN 10
#define MSG "Programme de démonstration\n"
```

Directives du préprocesseur :
accès avant la compilation

```
int fonc1(int x);
int fonc2(int x);
```

Déclaration des fonctions

```
void main()
```

```
{
    int i;
```

/ début du bloc de la fonction main */*
/ définition des variables locales */*

```
    i = 0 ;
    fonc1(i) ;
    fonc2(i) ;
```

```
}
```

/ fin du bloc de la fonction main */*

Programme
principal

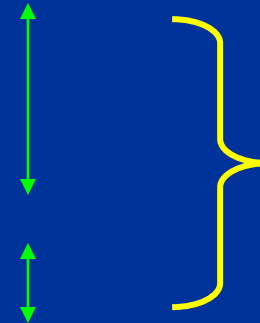
```
int fonc1(int x) {
    return x;
}
```

Définitions des
fonctions

```
int fonc2(int x) {
    return (x * x);
}
```

Structure d'un programme C

```
#include <stdio.h>
#define DEBUT -10
#define FIN 10
#define MSG "Programme de démonstration\n"
int fonc1(int x);
int fonc2(int x);
```



Fichier.h

```
#include "Fichier.h"
```

```
void main()
```

```
{
```

```
    int i;
```

```
    i = 0 ;
```

```
    fonc1(i) ;
```

```
    fonc2(i) ;
```

```
}
```

```
int fonc1(int x) {
```

```
    return x;
```

```
}
```

```
int fonc2(int x) {
```

```
    return (x * x);
```

```
}
```

```
/* début du bloc de la fonction main */
/* définition des variables locales */
```

```
/* fin du bloc de la fonction main */
```



Programme
principal

Définitions des
fonctions

MANIPULATION DES FICHIERS

Règles de visibilité des variables

Les variables ne peuvent être utilisées que là où elles sont déclarées

```
double func(int x);
int glob=0;           // variable globale

void main(void)
{
    int i = 5, j, k = 2; //locales à main
    glob++;
    func (i);
    func (k);
}

double func(int v)
{
    double d, f=v;      //locales à func

    glob++;
    f += glob;
    return f;
}
```

```
#include <stdio.h>

int next(){
    static value = 0;
    return value++;
}

void main(void) {
    printf("Appel 1 : %d\n",next());
    printf("Appel 2 : %d\n",next());
}
```

Appel 1 : 0

Appel 2 : 1

Variables et fonctions externes

Le fichier impose lui aussi un domaine de visibilité. Une variable définie globale au fichier (en dehors de toute fonction) ne sera alors visible que par les fonctions de ce fichier. Le problème est de savoir comment exporter cette variable pour d'autres fonctions du programme (externes au module) si le besoin s'en fait ressentir ?

Fichier "Module.h"	Fichier "Module.c"
<pre>extern int a; void fct();</pre>	<pre>#include "module.h" int a = 0; void fct() { a++; }</pre>
Fichier "main.c"	
<pre>#include <stdio.h> #include "module.h" void main(void) { fct(); a++; printf("%d\n",a); }</pre>	

Directives de compilation

Les directives de compilation permettent d'inclure ou d'exclure du programme des portions de texte selon l'évaluation de la condition de compilation.

<code>#if defined (symbole)</code>	<code>/* inclusion si symbole est défini */</code>
<code>#ifdef (symbole)</code>	<code>/* idem que #if defined */</code>
<code>#ifndef (symbole)</code>	<code>/* inclusion si symbole non défini */</code>
<code>#if (condition)</code>	<code>/* inclusion si condition vérifiée */</code>
<code>#else</code>	<code>/* sinon */</code>
<code>#elif</code>	<code>/* else if */</code>
<code>#endif</code>	<code>/* fin de si */</code>
<code>#undef symbole</code>	<code>/* supprime une définition */</code>

Exemple :

```
#ifndef (BOOL)
#define BOOL char      /* type boolean */
#endif
```

```
#ifdef (BOOL)
    BOOL FALSE = 0;    /* type boolean */
    BOOL TRUE = 1;     /* définis comme des variables */
#else
#define FALSE 0        /* définis comme des macros */
#define TRUE 1
#endif
```

Utile pour les fichiers include.

```
#ifndef _STDIO_H_
#define _STDIO_H_
    texte a compiler une fois
    ...
#endif
```

Listes chaînées

Une liste est une structure de données constituée de cellules chaînées les unes aux autres par pointeurs.

Une liste simplement chaînée : une cellule est un enregistrement qui peut être déclarée comme suit:

```
struct Node {  
    int          data;          /* les informations */  
    struct Node  *next;         /* le lien */  
};
```

Une liste doublement chaînée

```
struct Node {  
    int          data;          /* les informations */  
    struct Node  *next;         /* lien vers le suivant */  
    struct Node  *prev;         /* lien vers le précédent */  
};
```

Structures de données dynamiques

typedef : mot réservé,
crée de nouveaux noms
de types de données

Ex :

```
typedef char *  
STRING;
```

fait de STRING un synonyme
de "char * «

Portée : comme les variables.
p=new struct Node;

```
typedef struct Node {  
    int      data;  
    struct Node *next;  
}cellule;
```

```
cellule * new_node(int value)  
{  
    cellule * p;
```

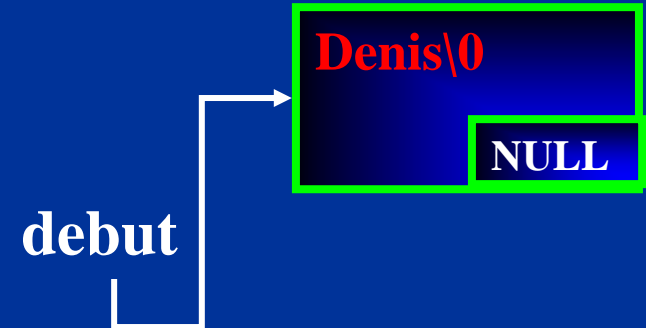
```
    p = (cellule *)malloc(sizeof(cellule));  
    p->data = value;  
    p->next = NULL;
```

```
    return p;
```

```
}
```

Nouvelle cellule dans une liste chaînée vide

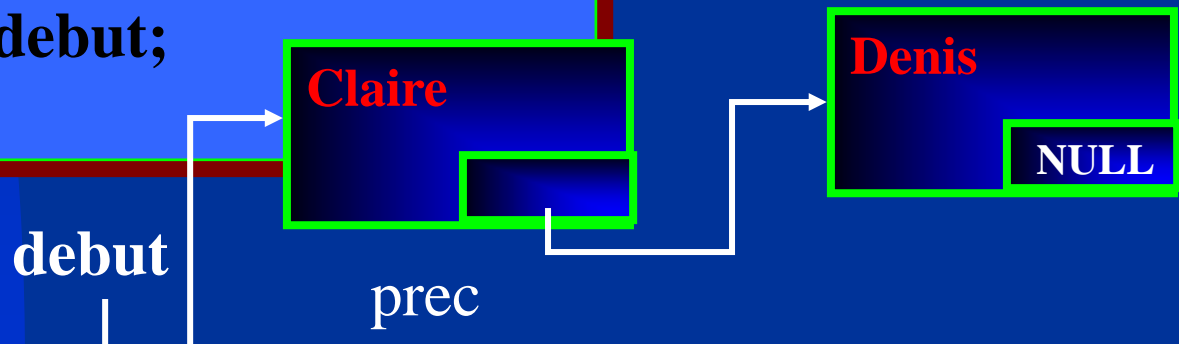
```
cellule *debut;  
debut = (cellule *) malloc (sizeof(cellule));  
strcpy ( debut->name, "Denis");  
debut->next = NULL;
```



Le début de la liste est indiqué par un pointeur indépendant (debut) et la fin par NULL

Nouvelle cellule en début de liste chaînée

```
cellule *prec;  
prec = (cellule *) malloc (sizeof(cellule));  
strcpy ( prec->name, "Claire");  
prec->next = debut;  
debut = prec;
```

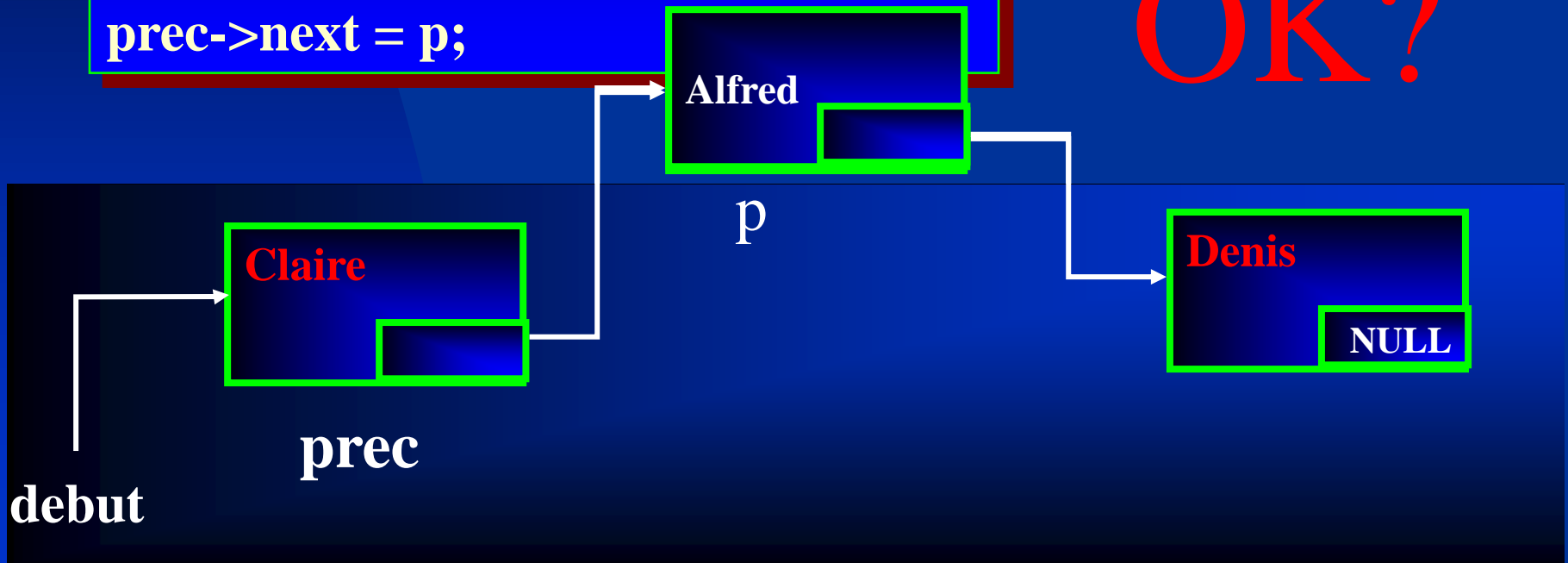


Nouvelle cellule après la cellule prec

```
cellule *p;  
p = (cellule *) malloc (sizeof(cellule));  
strcpy ( p->name, "Alfred");  
p->next = prec->next;  
prec->next = p;
```



OK?

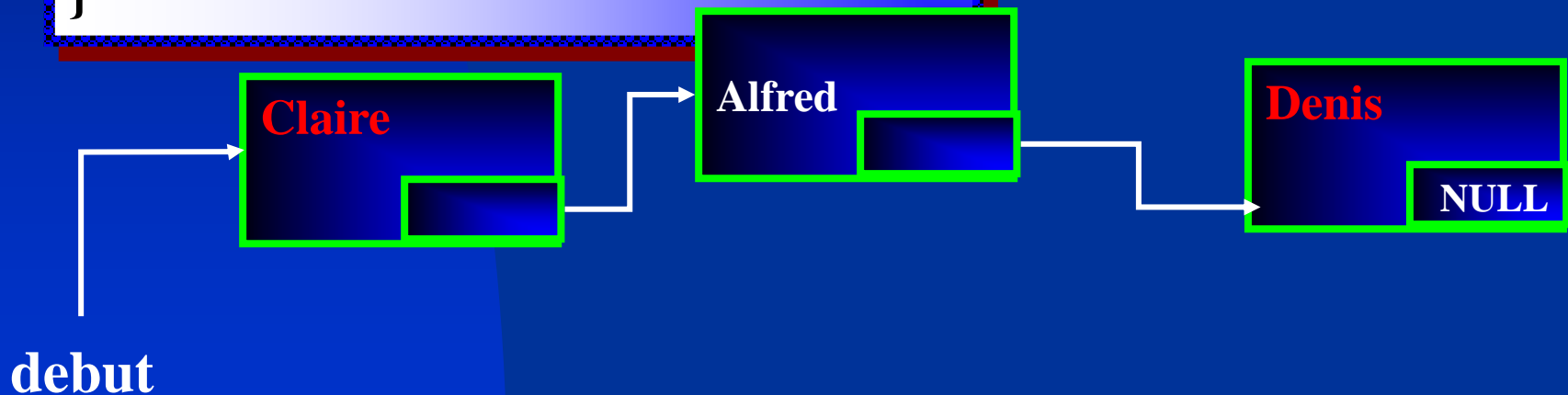


Parcourir une liste

```
void parcours (struct Node *debut)
{
    struct Node *p;
    p = debut;
    while ( p != NULL) {
        printf ("%s\n", p->name);
        p = p->next;
    }
}
```

debut est un pointeur sur la cellule qui contient le premier élément de la liste

Liste identifier par l'adresse de sa première cellule



```
void liberation(liste L){  
    if (L) {  
        liste temp = L-  
            >suivant;  
        free(L);  
        liberation(temp);  
    }  
}
```

```
void liberation (liste *lis)  
{  
    liste *p;  
    while ( lis != NULL) {  
        p = lis;  
        lis=lis->next;  
        free(p);  
    }  
}
```

Exemple

LIFO

Last-In-First-Out

- *Le dernier élément ajouté dans la liste, est le premier à en sortir.*
- *Opérations:*
 - Créer la pile,*
 - Ajouter un élément (Push),*
 - Effacer un élément (Pop),*
 - Eliminer la pile .*

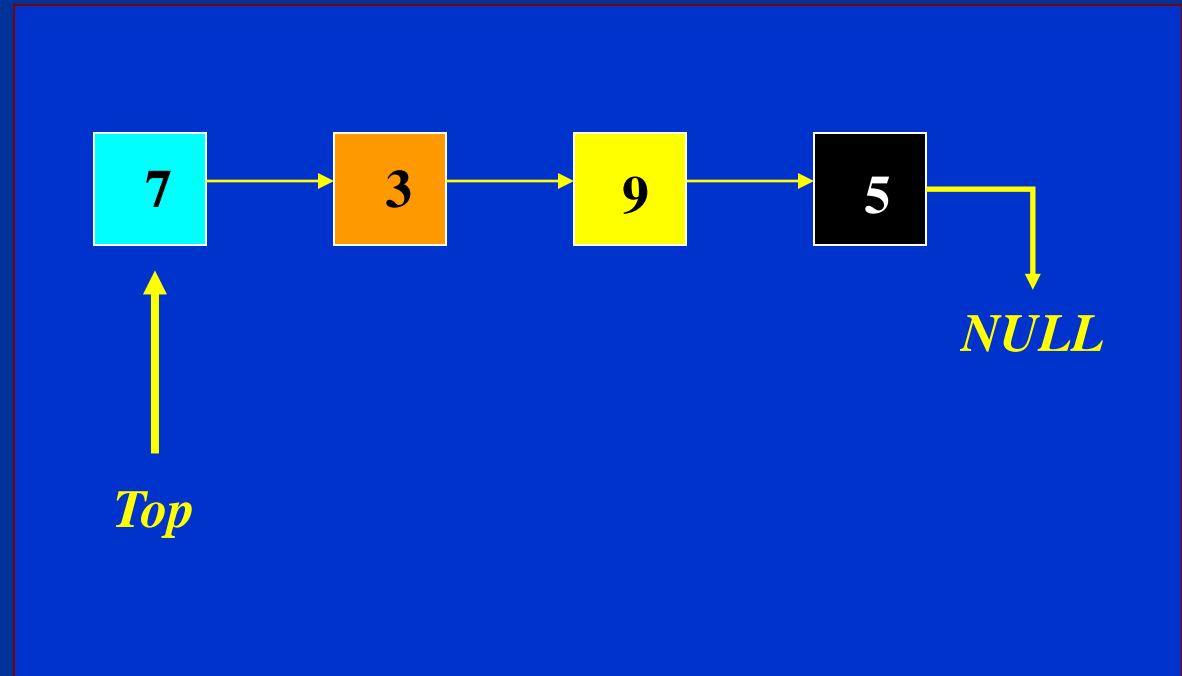
PUSH procédure

Push(5)

Push(9)

Push(3)

Push(7)



Overflow risk si la pile est pleine!!!

POP procédure

Pop(7)

Pop(3)

Pop(9)

Pop(5)

*Underflow risk
si la pile est vide !!!*

Simulation de la factorielle

$$5! = ??$$

$$5! = 5 * 4!$$

$$4! = 4 * 3!$$

$$3! = 3 * 2!$$

$$2! = 2 * 1!$$

$$1! = 1 * 0!$$

$$0! = 1$$

0
1
2
3
4
5

Simulation de la factorielle

$$5! = ??$$

$$5! = 5 * 4!$$

$$4! = 4 * 3!$$

$$3! = 3 * 2!$$

$$2! = 2 * 1!$$

$$1! = 1 * 0!$$

$$0! = 1$$

$$0! = 1$$

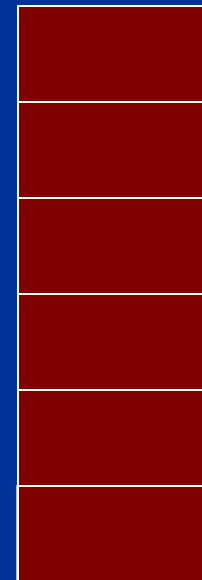
$$1! = \boxed{1} * 1 = 1$$

$$2! = \boxed{2} * 1 = 2$$

$$3! = \boxed{3} * 2 = 6$$

$$4! = \boxed{4} * 6 = 24$$

$$5! = \boxed{5} * 24 = 120$$



TP3

Créer une liste simplement chaînée dont chaque cellule contient les champs suivants :

```
{  
    int          data;    /* les informations */  
    struct Node  *next;   /* le lien : pointeur sur la cellule suivante */  
}
```

Le programme doit gérer en boucle le menu de choix suivant :

- 1- Créer une liste avec insertion en tête de liste (avec affichage)
- 2- Sauver la liste
- 3- Insérer dans la liste (avec affichage)
- 4- Nettoyer la mémoire et sortir

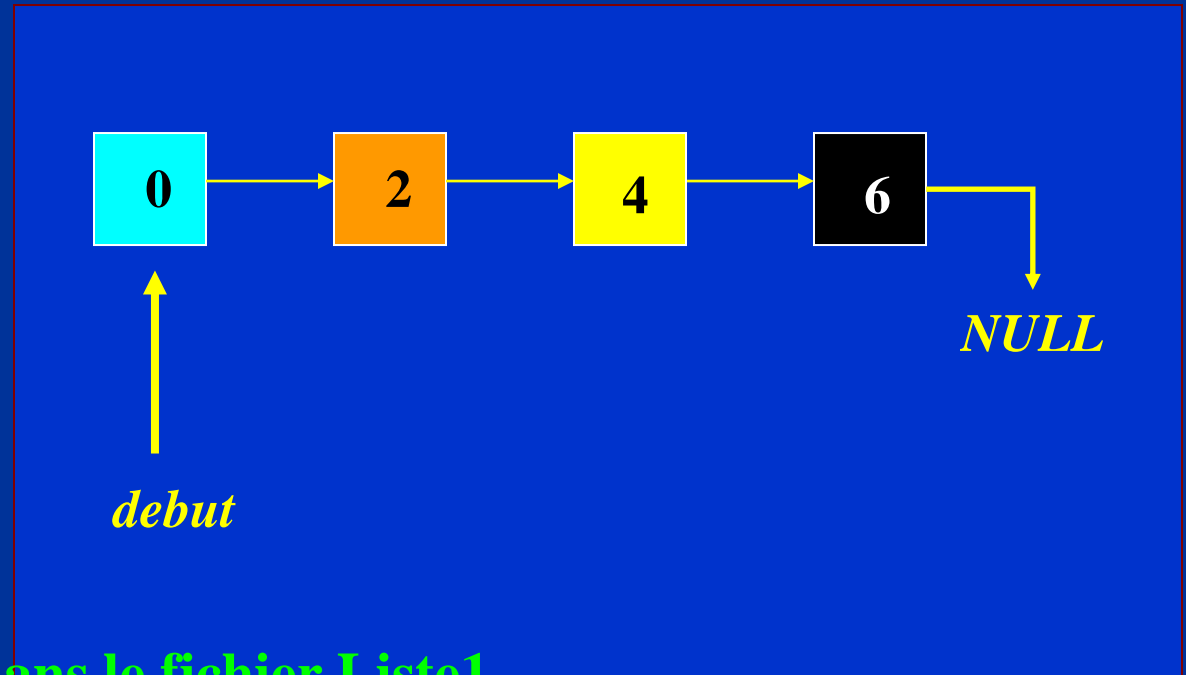
1 : Crée une liste en insérant $2n$, puis $2n-2$, puis $2n-4$, ... et enfin 0 qui sera en tête de la liste. (Liste : 0,2,4,6,8,10, ... $2n$.)

Push(6)

Push(4)

Push(2)

Push(0)



2 : Sauvegarde la liste dans le fichier Liste1

3 : Insère dans la liste -1 , puis 1 , puis 3 , ... et en fin $2n-1$.

(Liste finale : $-1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \dots, 2n$)

2' : Sauvegarde la nouvelle liste dans le fichier Liste2

4 : Libère la mémoire avant de quitter le programme.

Listes doublement chaînées

```
struct Node {  
    int          data;          /* les informations */  
    struct Node  *prev;         /* lien vers le précédent */  
    struct Node  *next;         /* lien vers le suivant */  
};
```



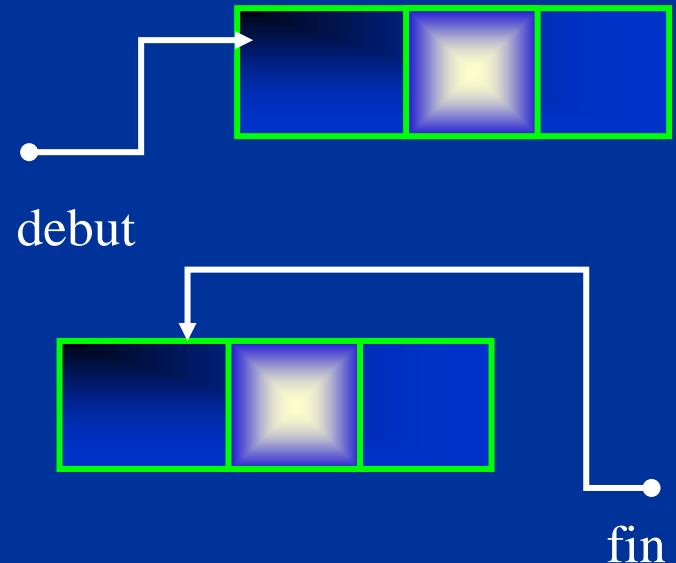
```
typedef struct Node Cell;
```

```
Cell *debut, *fin, *act;
```

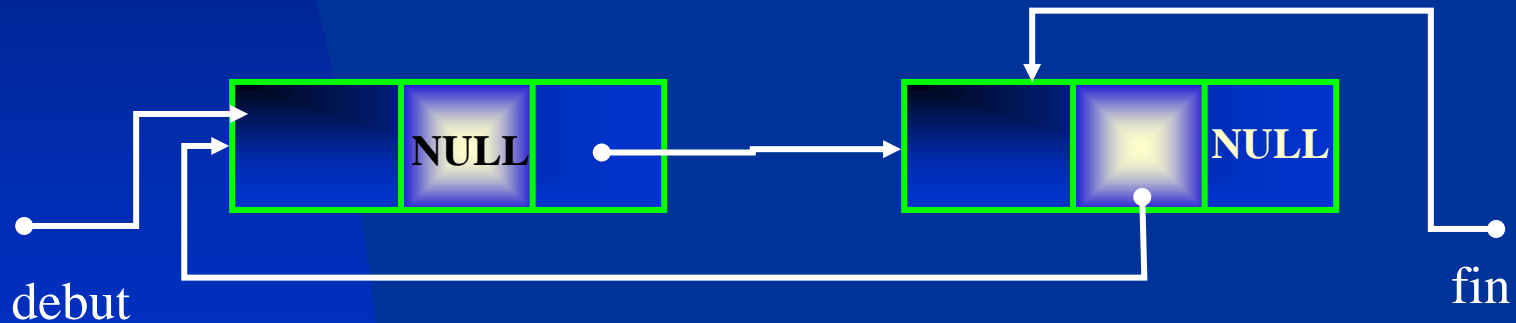
```
int dim = sizeof(Cell);
```

```
debut = (Cell *) malloc(dim);
```

```
fin   = (Cell *) malloc(dim);
```

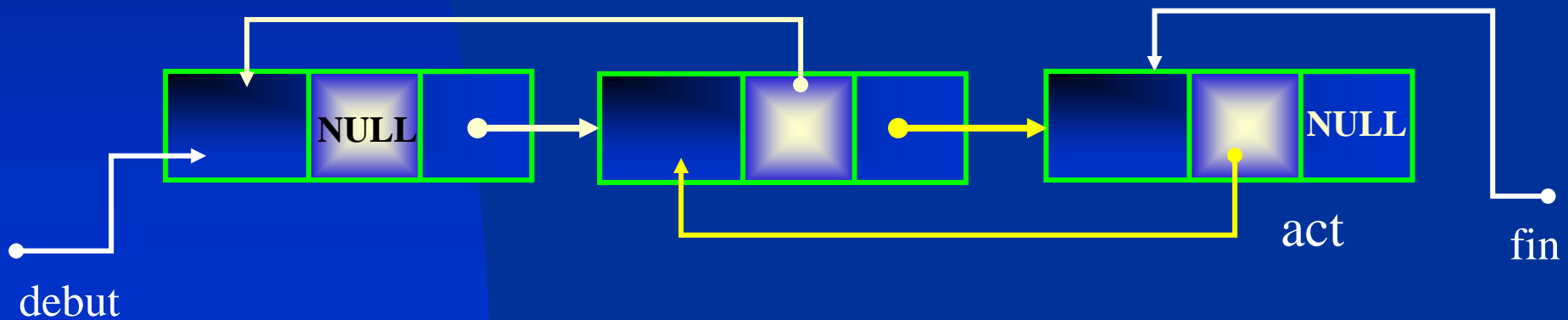
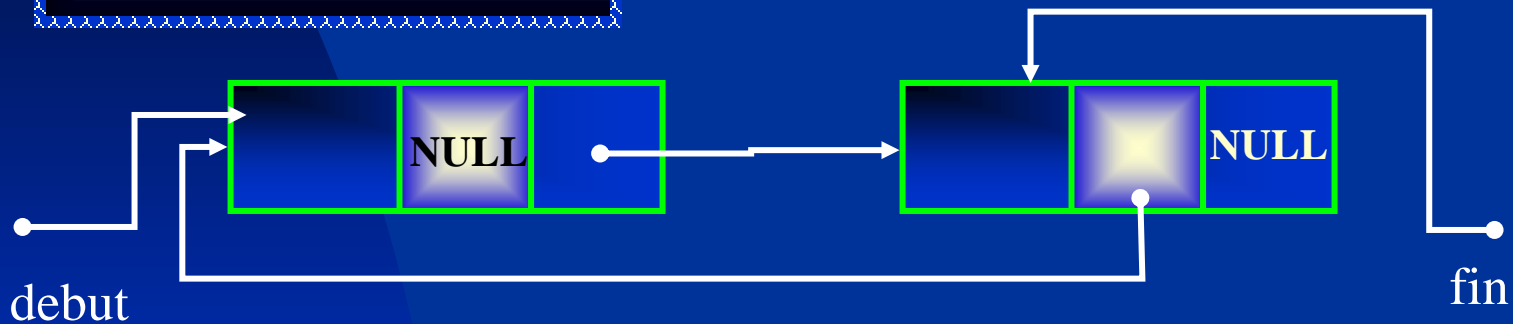


```
debut->prev = NULL;  
debut->next = fin;  
fin->prev = debut;  
fin->next = NULL;
```



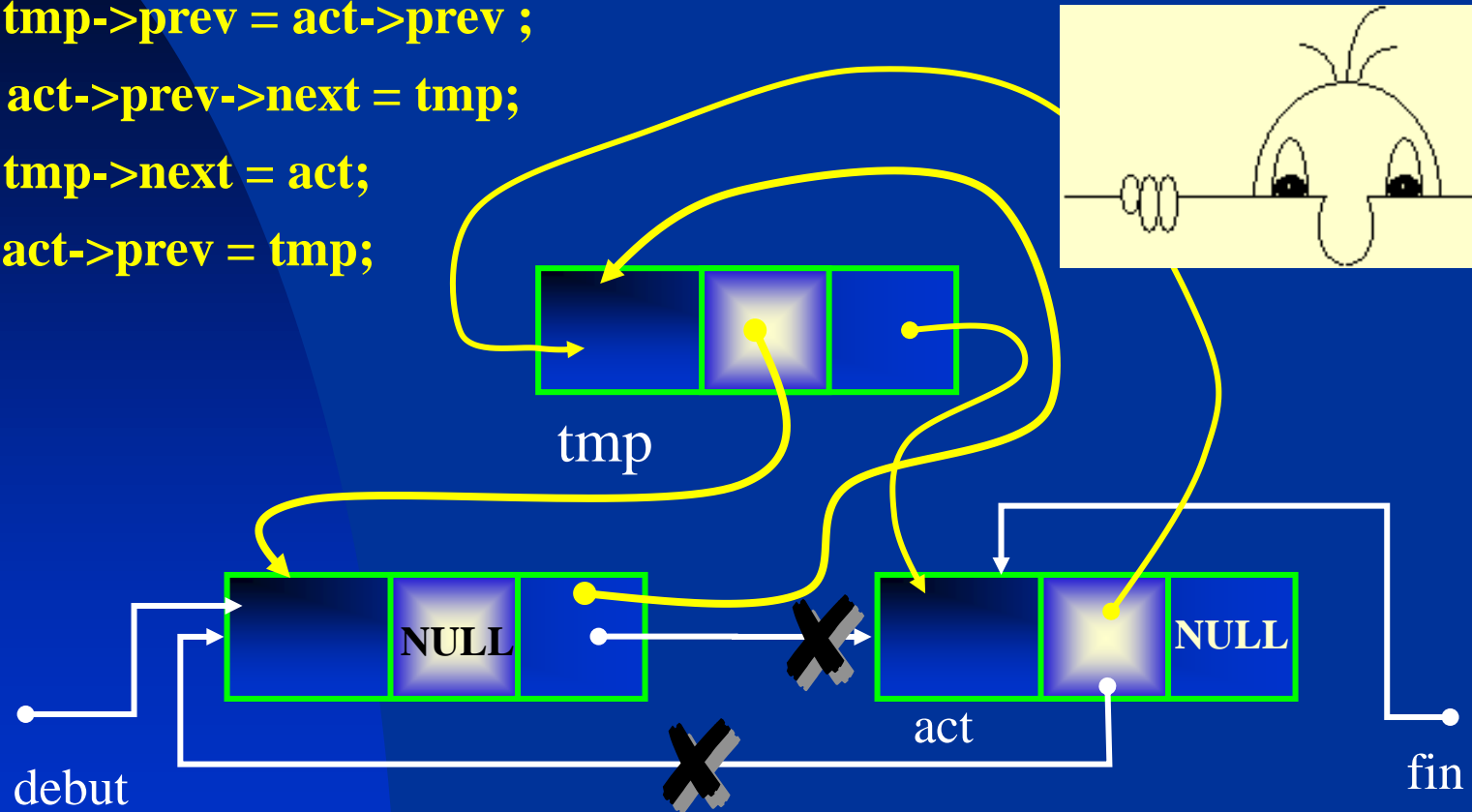
Insérer un élément en fin de liste

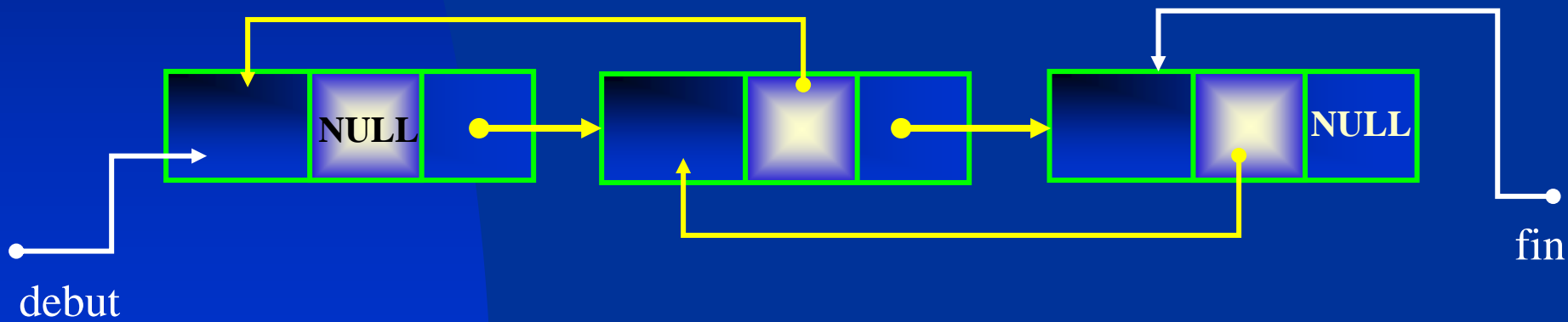
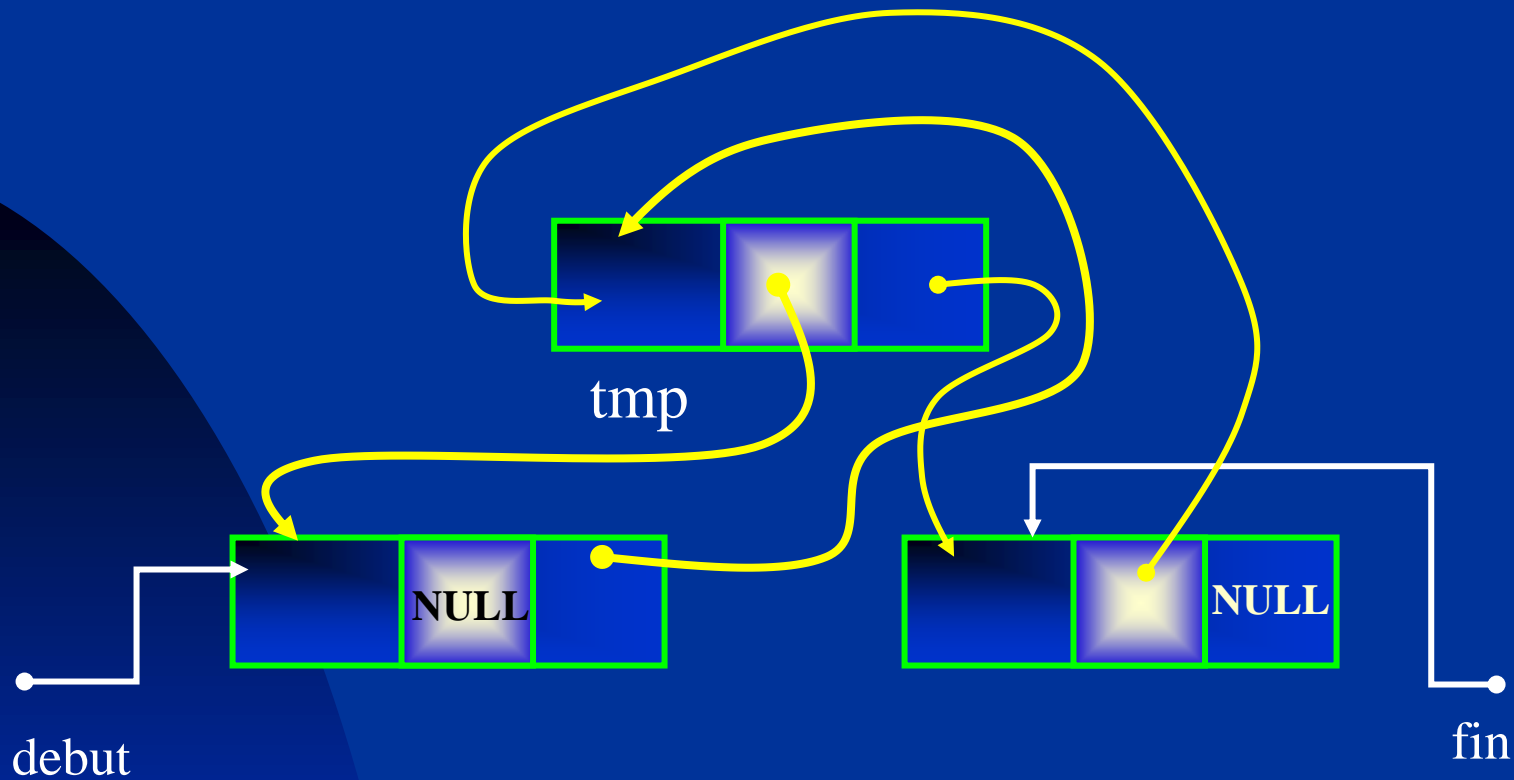
```
fin->next = act;  
act->prev = fin;  
act->next = NULL;  
fin = act;
```



Insérer un élément dans la liste pos=qlq

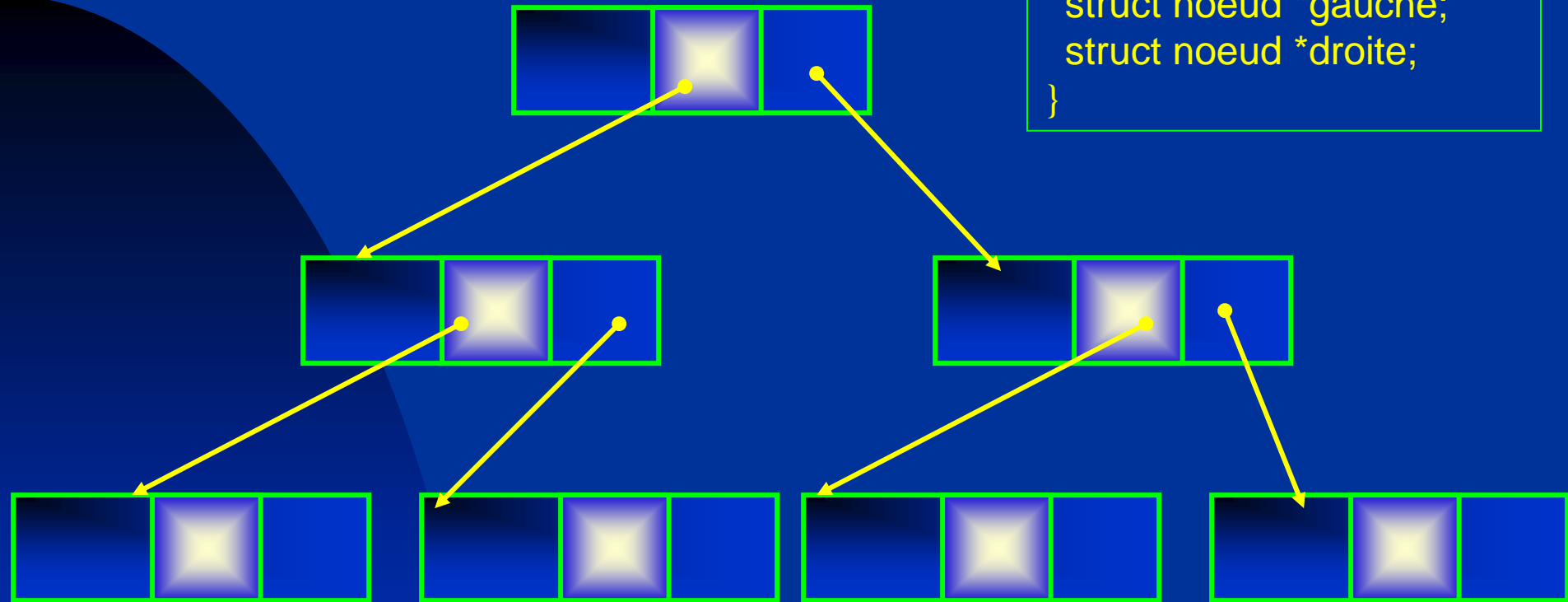
```
Cell *tmp;  
tmp = (Cell *) malloc(dim);  
tmp->prev = act->prev ;  
act->prev->next = tmp;  
tmp->next = act;  
act->prev = tmp;
```





Arbre binaire

```
typedef struct noeud {  
    int donnee;  
    struct noeud *gauche;  
    struct noeud *droite;  
}
```



Tri par arbre binaire de recherche ABR