

Programmation II

SMA-S4
FP Khouribga

Les Pointeurs en C

2

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

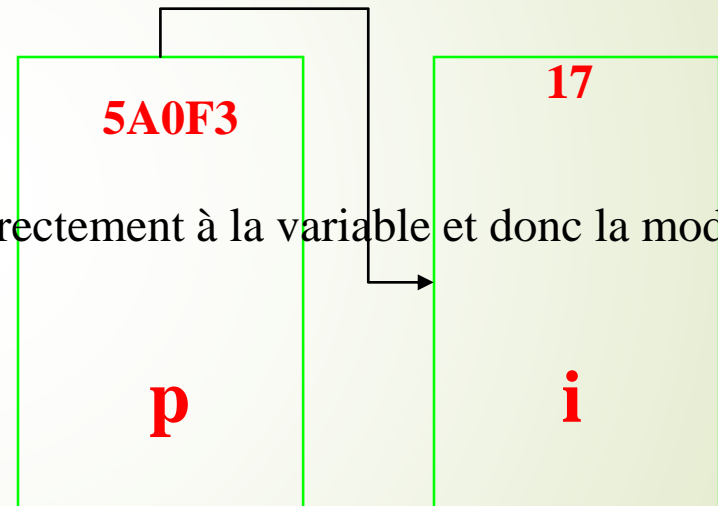
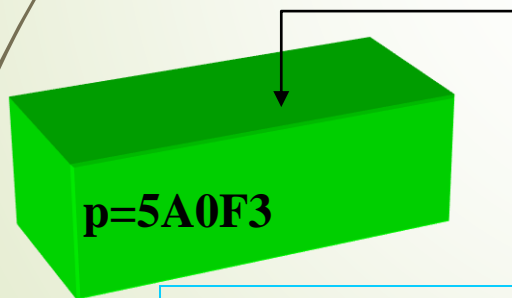
Les pointeurs, c'est quoi?

3

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 ?

4

- 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

5

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)

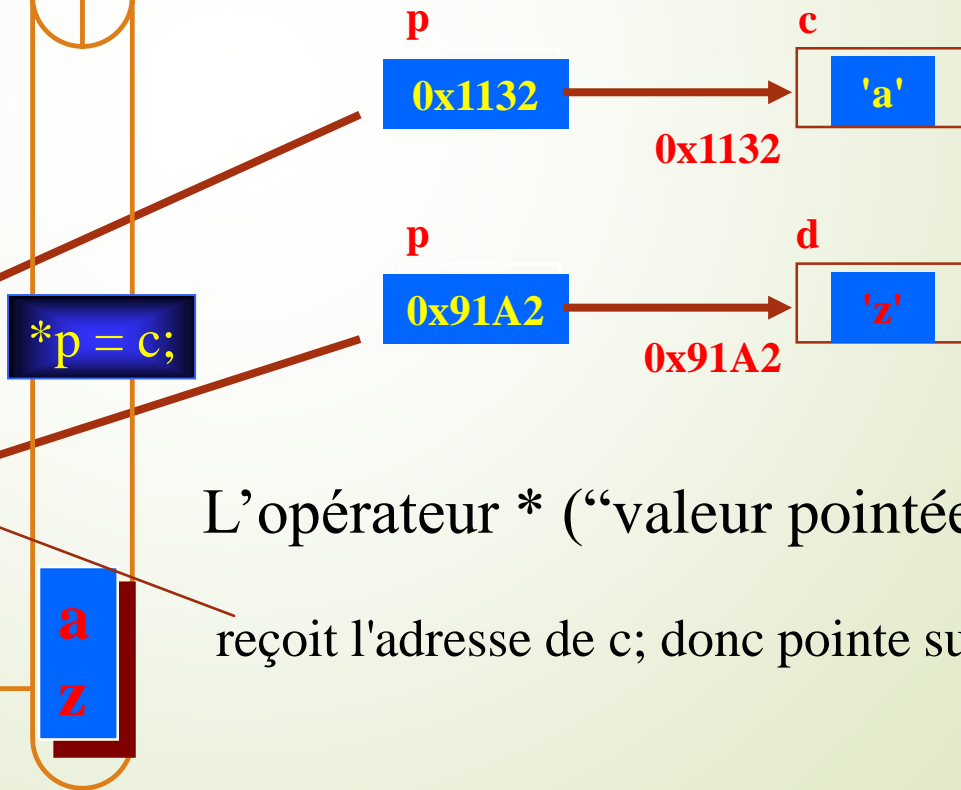
6

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



L'opérateur * ("valeur pointée par")

reçoit l'adresse de c; donc pointe sur c.

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

```
int 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 ?

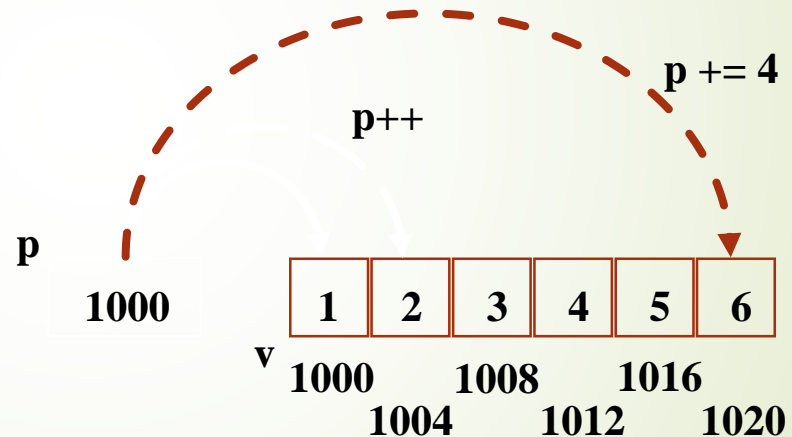
```
}
```

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++) passer à l'adresse suivante

*(p++) trouver valeur de nouvellement pointée

*++p signifie:

*++p incrémenter d'abord le pointeur

*++p trouver la valeur pointée



Passage des paramètres par valeur et par adresse

10

Syntaxe qui conduit à une erreur:

```
#include <stdio.h>

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

void main()
{
    int a = 5 , b = 8;
    echange(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
echange 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 echange(int *x,int *y)
{
    int tampon;
    tampon = *x;
    *x = *y;
    *y = tampon;
}

void main()
{
    int a = 5 , b = 8 ;
    echange(&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

12

```
#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

14

- 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)
{
```

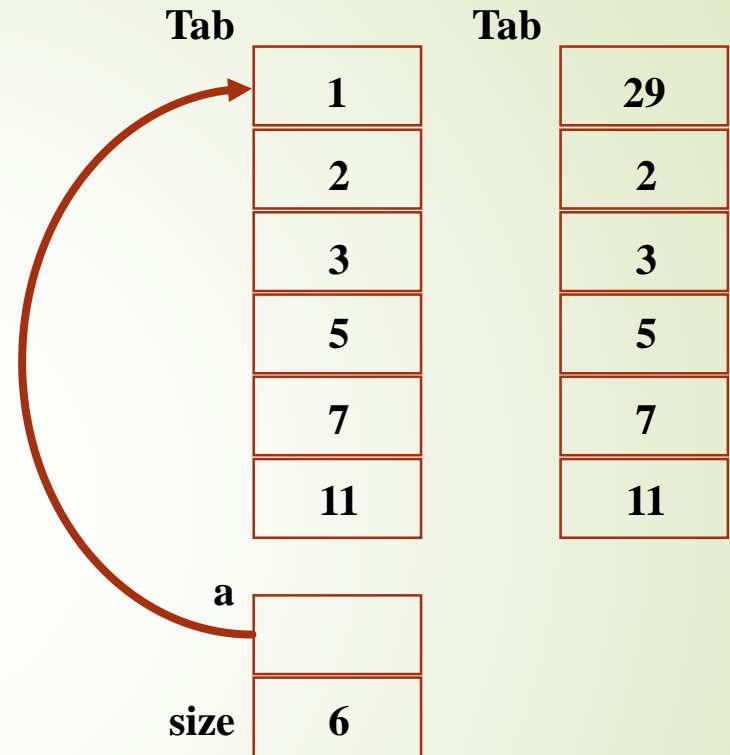
```
int add_elements(int *p, int size)
{
```

Exemple

15

```
#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

16

```
#include <stdio.h>

long sum(long*, int);

int main(void)
{
    long Tab[6] = { 1, 2,
                    3, 5, 7, 11 };

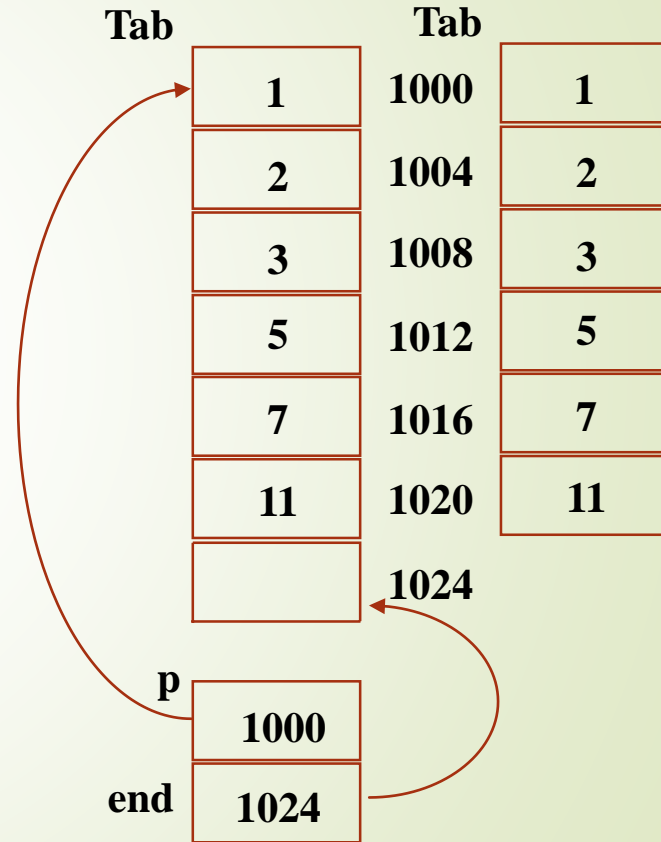
    printf("%ld\n", sum(Tab, 6));

    return 0;
}

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

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

    return total;
}
```



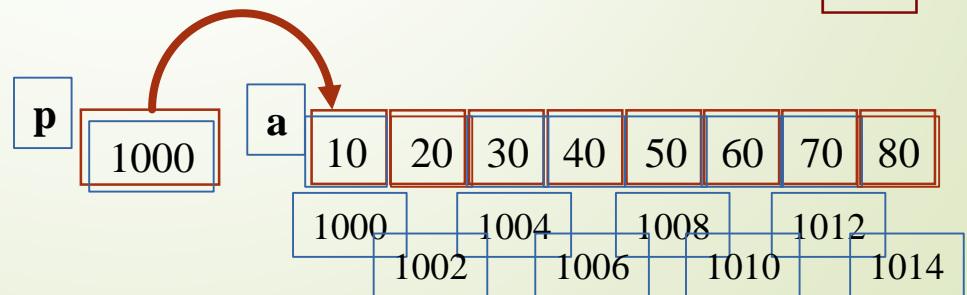
Quelle notation?

- $A[0]$ 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

18

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\n",adr1,adr2,*adr1,*adr2);
    free(adr1); // Libération de la mémoire
    free(adr2);
    free(texte);
```

```
}
```

sizeof

20

```
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

21

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

int 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 ( )  $\neq 0$ 
    {
        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);

23

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

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

28

```
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);
```

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 */
```

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

35

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

<i>0</i>	<i>1</i>	<i>2</i>	<i>3</i>	<i>4</i>	<i>5</i>	<i>6</i>	j
2	6	4	8	12	13	0	tab[j]

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

tab[j] < tab[j+1]

1 3	1 2	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

37

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>  
int 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

39

```
#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

40

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

41

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

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

void main()
{
    /* début du bloc de la fonction main*/
    /* définition des variables locales */
    int i;

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

/* fin du bloc de la fonction main */

int fonc1(int x) {
    return x;
}

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

Directives du préprocesseur :
accès avant la compilation

Déclaration des fonctions

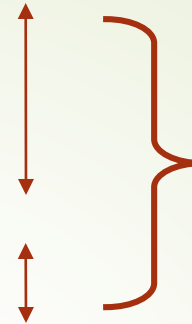
Programme
principal

Définitions des
fonctions

Structure d'un programme C

42

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

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

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

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

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

Programme
principal

Définitions des
fonctions

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

int 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

44

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" int main(void) { fct(); a++; printf("%d\n",a); }</pre>	

// Résultat affiché : 2

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 :

46

```
#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

47

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

48

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;
```

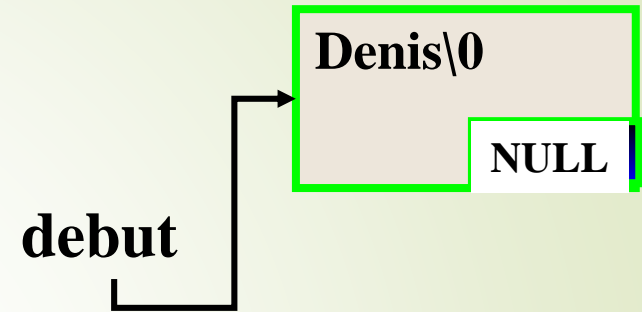
```
    return p;
```

```
}
```

Nouvelle cellule dans une liste chaînée vide

49

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

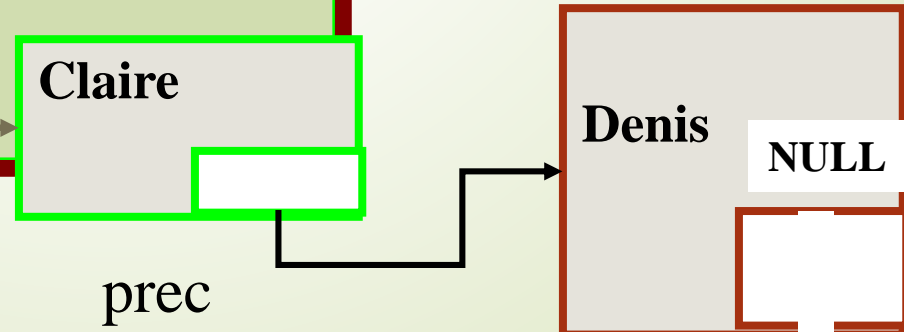


Le début de la liste est indiqué par un pointeur indépendant (debut) et la fin par NULL

Nouvelle cellule en début de liste chaînée

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

debut

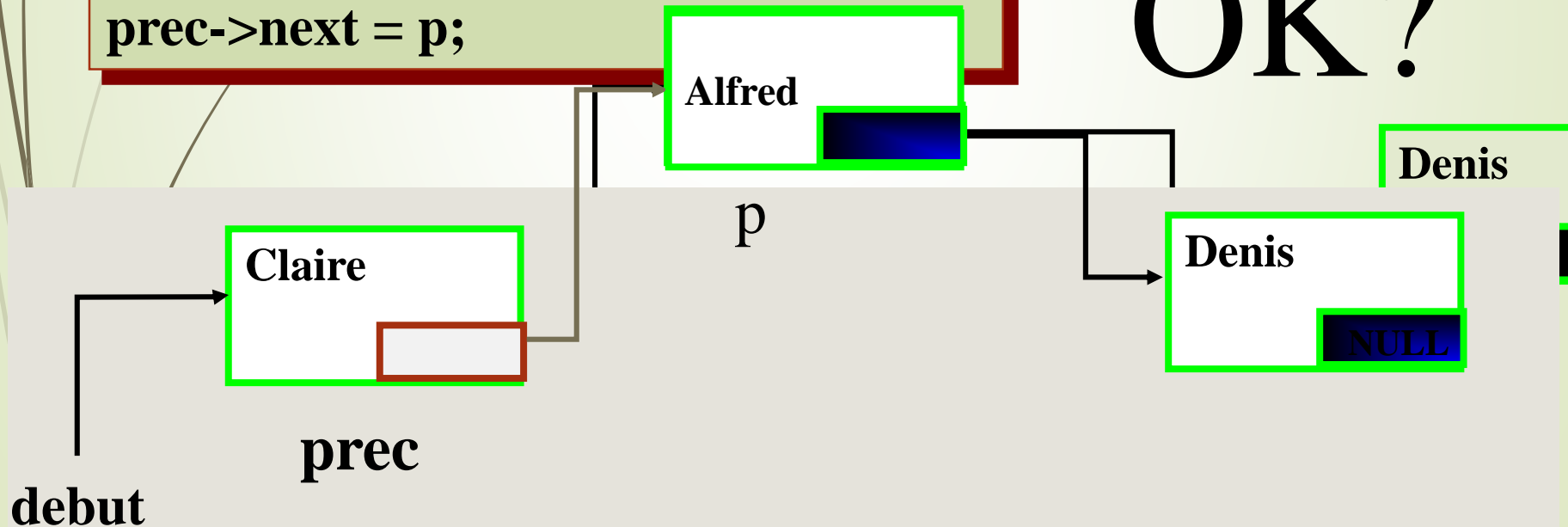


Nouvelle cellule après la cellule prec

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



OK?



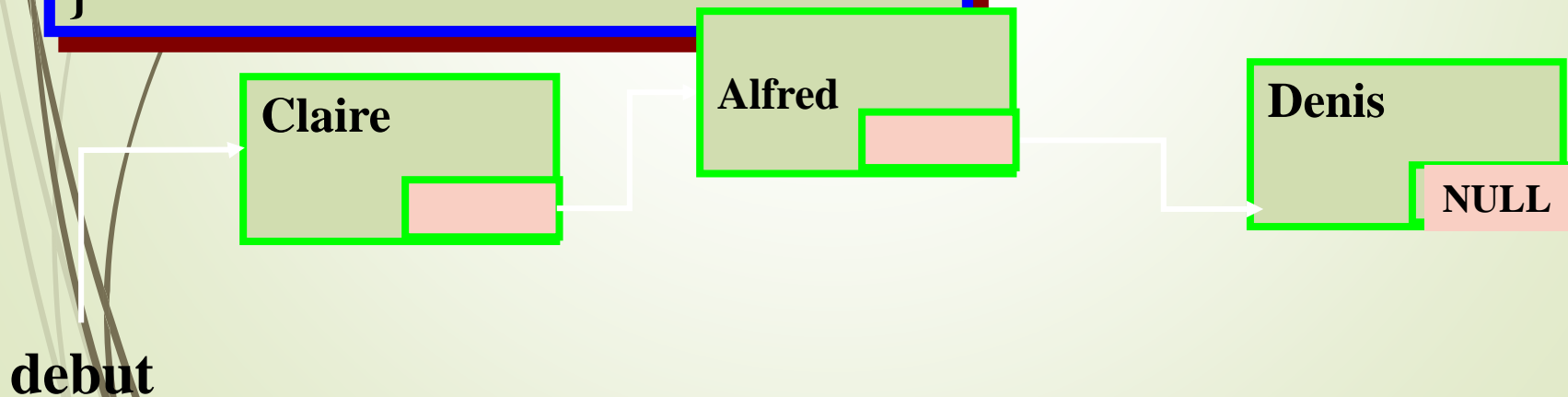
Parcourir une liste

51

```
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);  
    }  
}
```

LIFO

Exemple

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 .

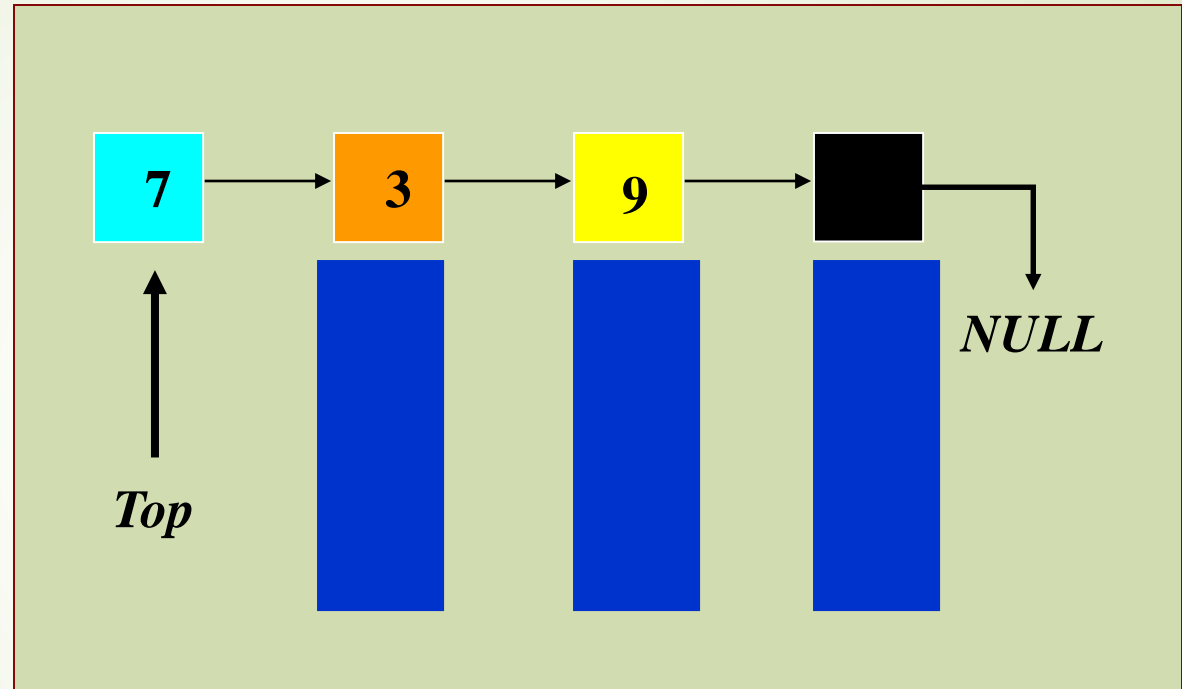
PUSH procédure

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

56

$$5! = ??$$

$$5! = 5 * 4!$$

$$4! = 4 * 3!$$

$$3! = 3 * 2!$$

$$2! = 2 * 1!$$

$$1! = 1 * 0!$$

$$0! = 1$$

0

1

2

3

4

5

Simulation de la factorielle

57

$$5! = ??$$

$$5! = 5 * 4!$$

$$4! = 4 * 3!$$

$$3! = 3 * 2!$$

$$2! = 2 * 1!$$

$$1! = 1 * 0!$$

$$0! = 1$$

$$0! = 1$$

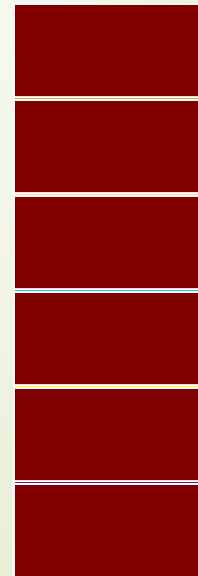
$$1! = 1 * 1 = 1$$

$$2! = 2 * 1 = 2$$

$$3! = 3 * 2 = 6$$

$$4! = 4 * 6 = 24$$

$$5! = 5 * 24 = 120$$



TP3

58

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

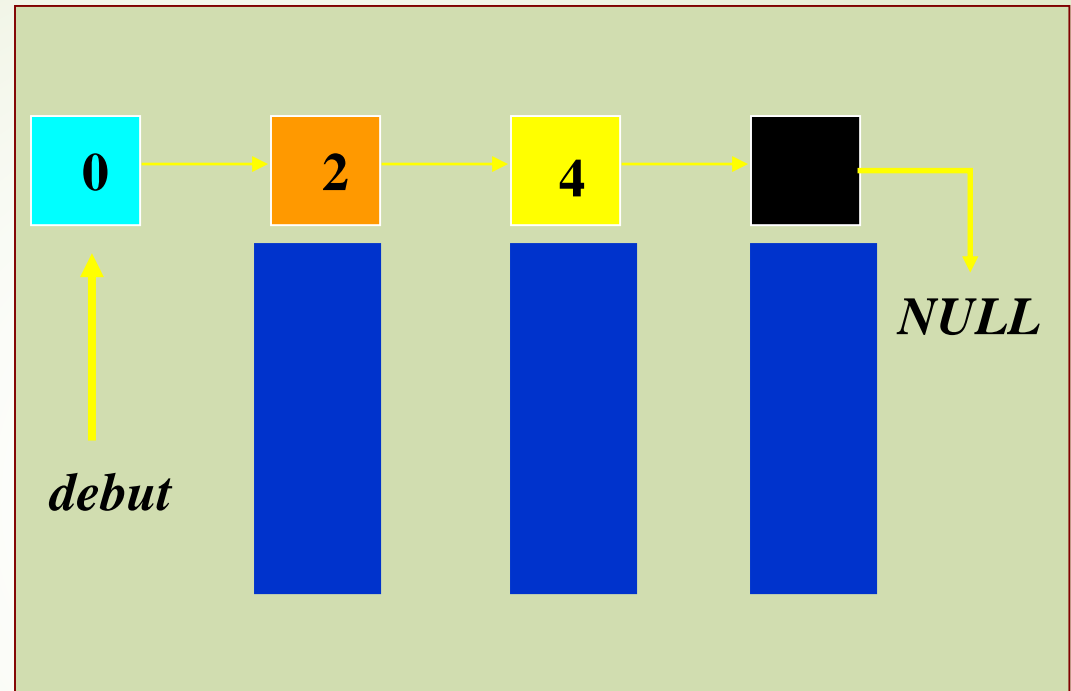
1 : Crée une liste en insérant $2n$, puis $2n-2$, puis $2n-4$, ... et enfin 0 qui sera en tête de la liste. (Liste : 0,2,4,6,8,10, ... $2n$.)

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 $2n-1$.

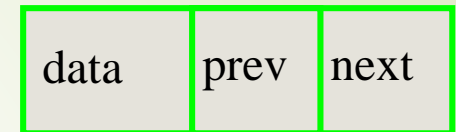
(Liste finale : $-1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \dots, 2n$)

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 */
};
```



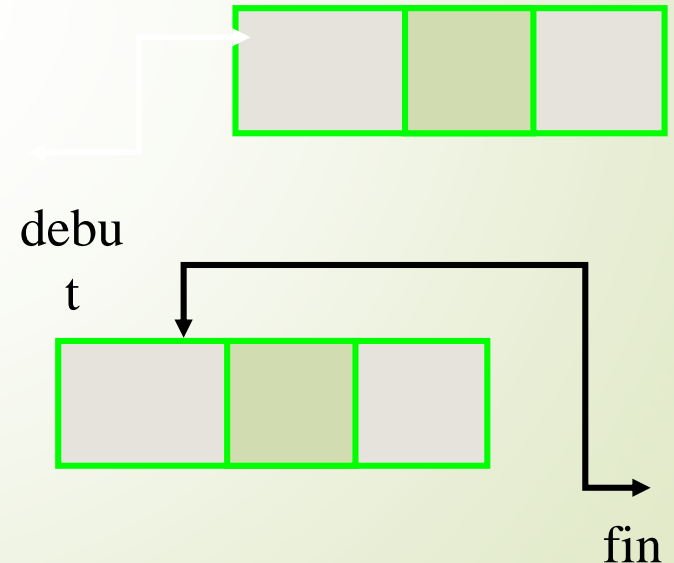
```
typedef struct Node Cell;
```

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

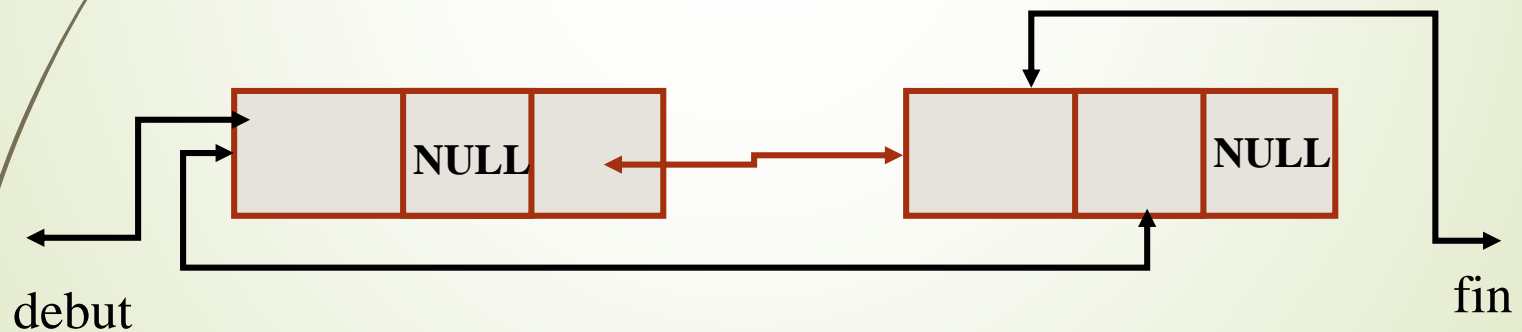
```
int dim = sizeof(Cell);
```

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

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



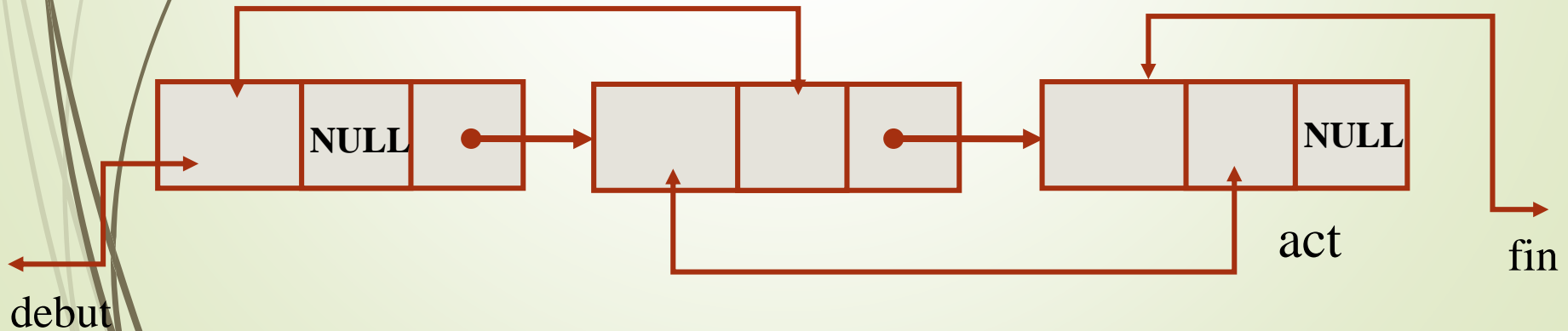
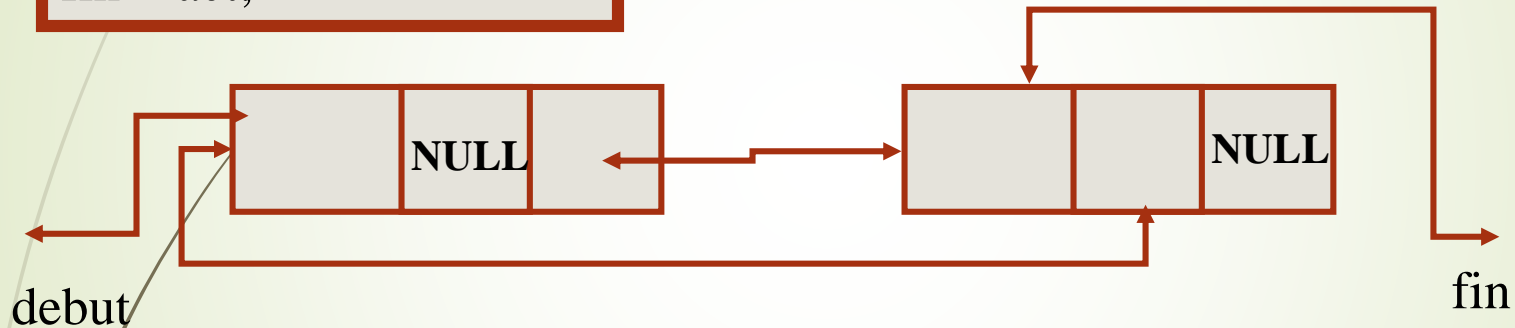
```
debut->prev = NULL;  
debut->next = fin;  
fin->prev = debut;  
fin->next = NULL;
```



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

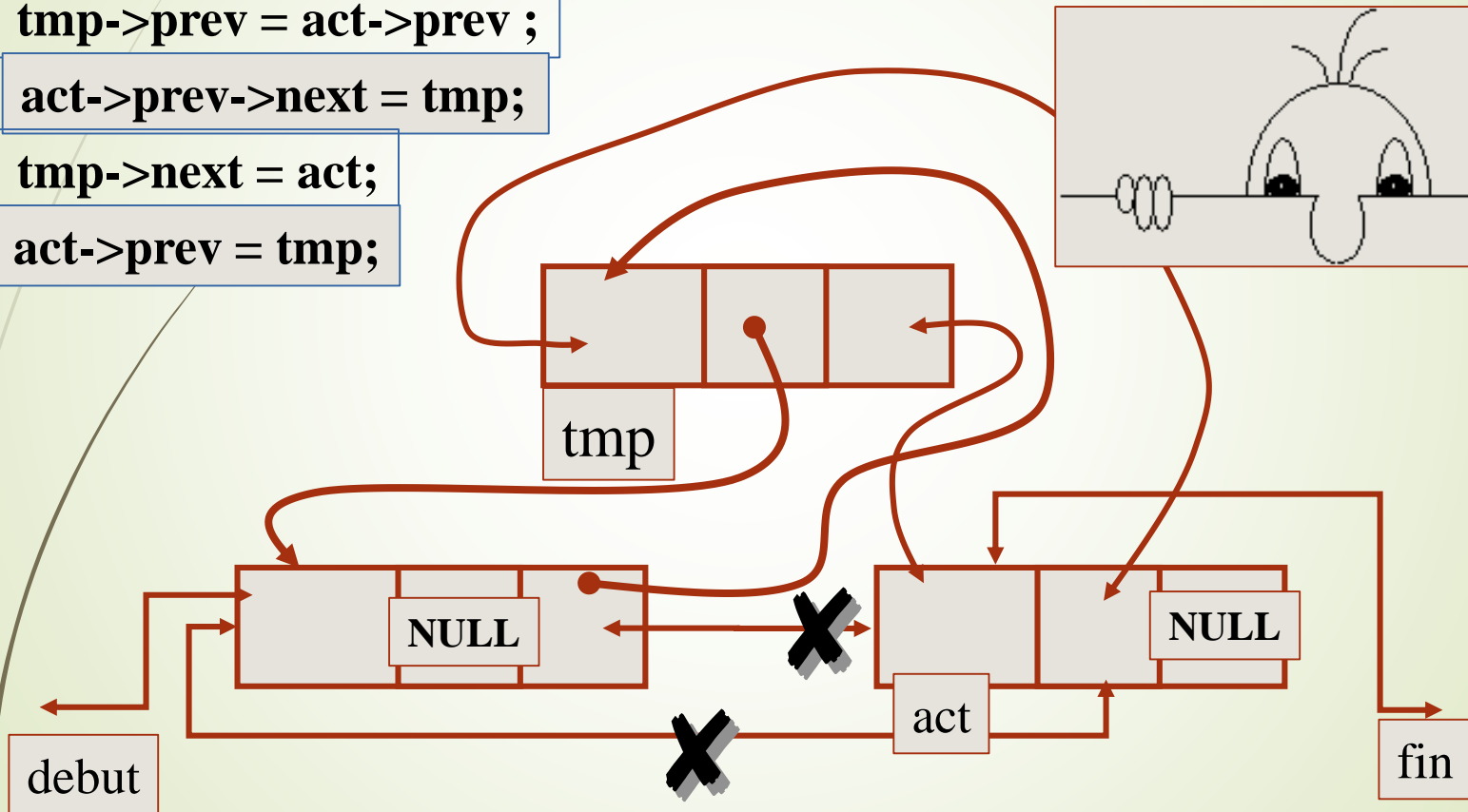
```
Cell *tmp;  
tmp = (Cell *) malloc(dim);
```

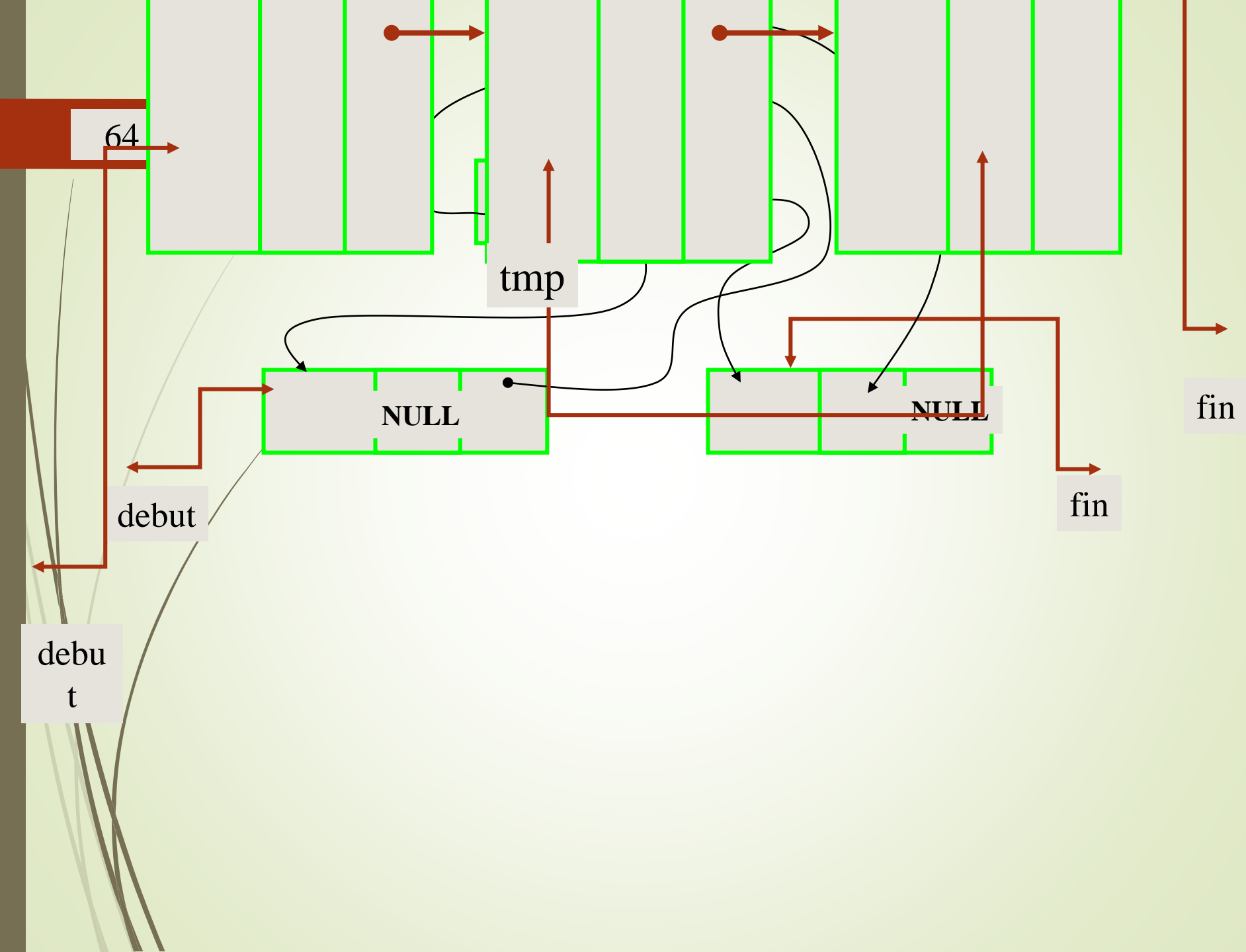
```
tmp->prev = act->prev ;
```

```
act->prev->next = tmp;
```

```
tmp->next = act;
```

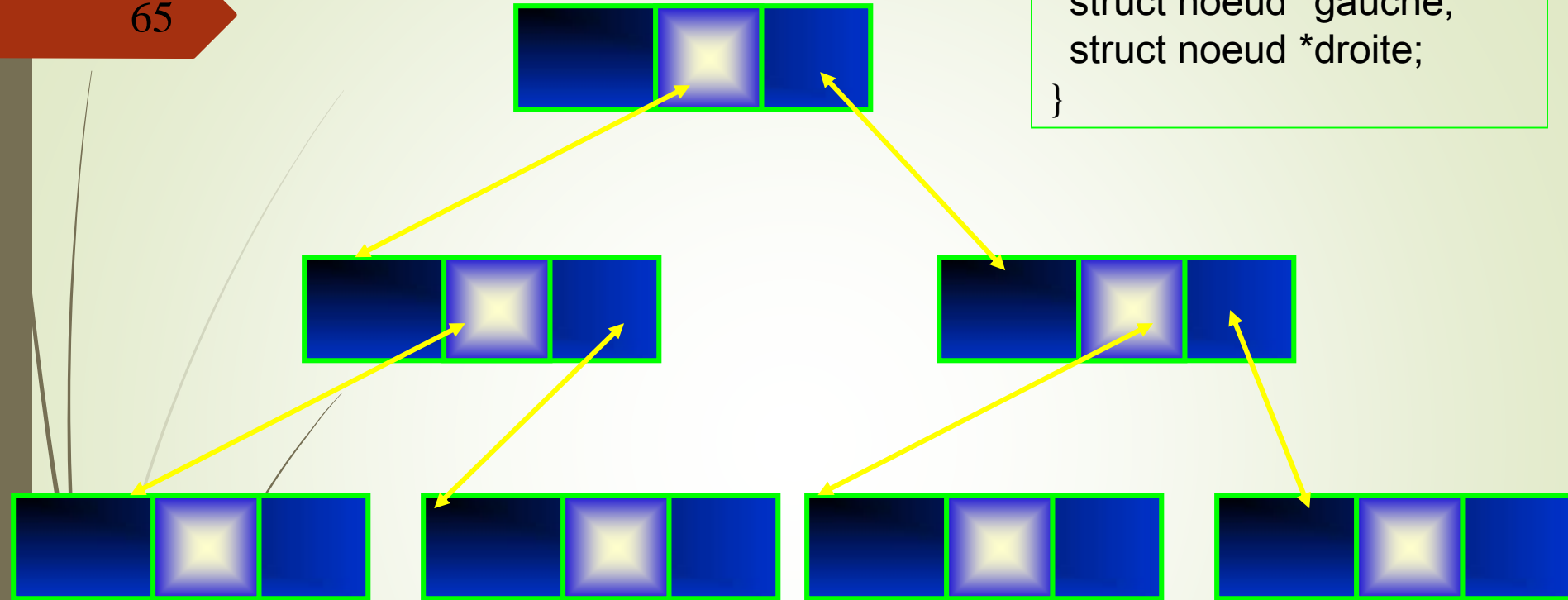
```
act->prev = tmp;
```





Arbre binaire

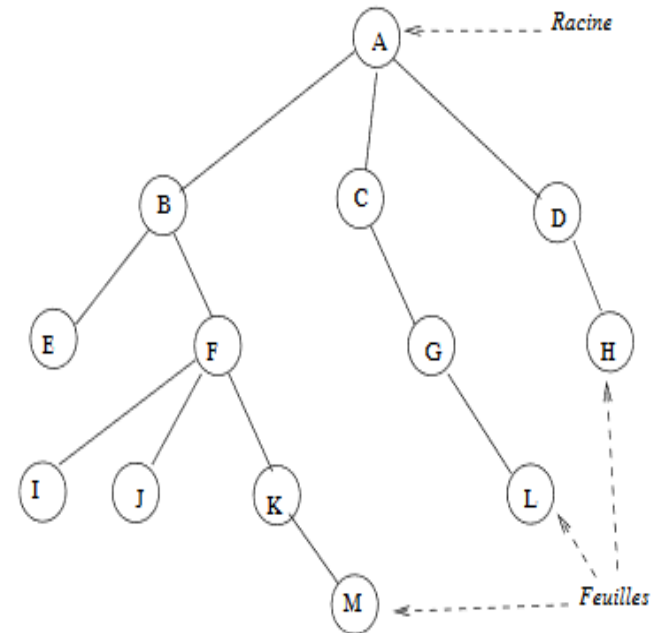
```
typedef struct noeud {  
    int donnee;  
    struct noeud *gauche;  
    struct noeud *droite;  
}
```



Tri par arbre binaire de recherche ABR

Arbre binaire de recherche

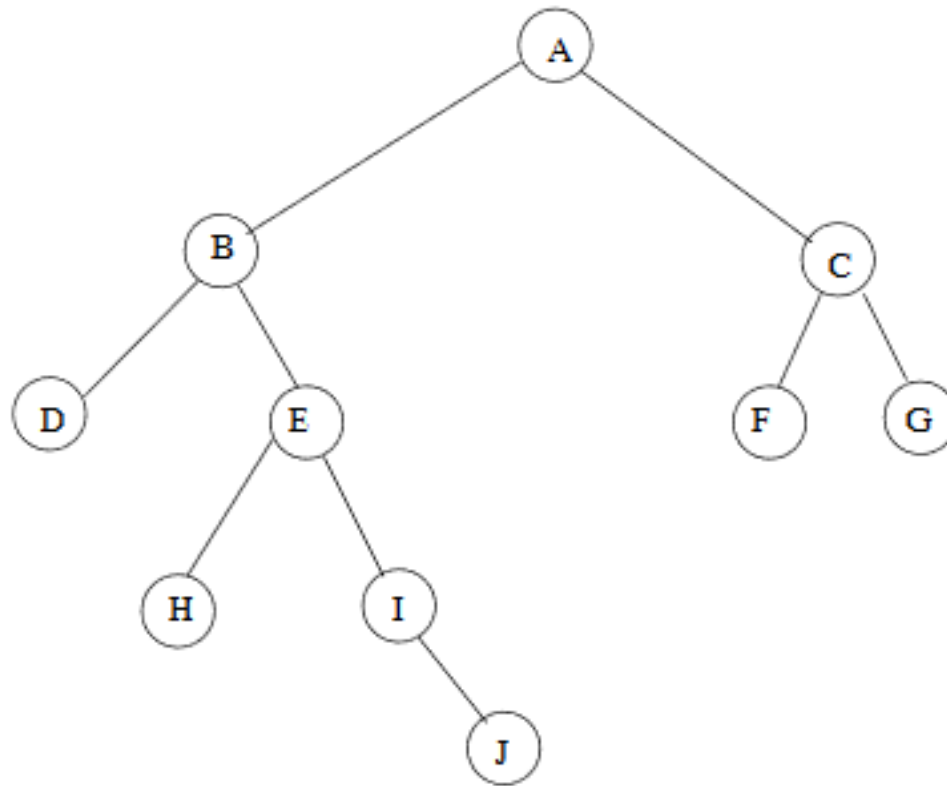
Un arbre est une structure composée de nœuds et de feuilles (nœuds terminaux) reliées par des branches. On le représente généralement en mettant la racine en haut et les feuilles en bas (contrairement à un arbre réel).



- Le nœud A est la racine de l'arbre.
- Les nœuds E, I, J, M, L et H sont des feuilles.
- Les nœuds B, C, D, F, G et K sont des nœuds intermédiaires.
- Si une branche relie un nœud n_i à un nœud n_j située plus bas, on dit que n_i est un ancêtre de n_j .
- Dans un arbre, un nœud n'a qu'un seul père (ancêtre direct).
- Un nœud peut contenir une ou plusieurs valeurs.
- La hauteur (ou profondeur) d'un nœud est la longueur du chemin qui le lie à la racine

Arbres binaires

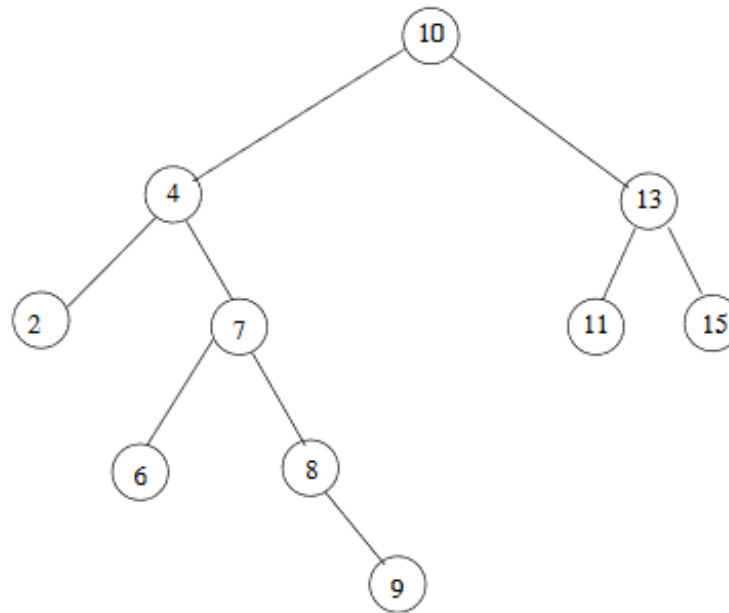
Un arbre binaire est un arbre tel que les nœuds ont au plus deux fils(gauche et droite).



Arbres binaires de recherche

Un arbre binaire de recherche est un arbre qui possède les propriétés suivantes:

- Tous les nœuds du sous-arbre de gauche d'un nœud de l'arbre ont une valeur inférieure ou égale à la sienne
- Tous les nœuds du sous-arbre de droite d'un nœud de l'arbre ont une valeur supérieure ou égale à la sienne



Recherche dans l'arbre

Un arbre binaire de recherche est fait pour faciliter la recherche d'informations.

La recherche d'un nœud particulier de l'arbre peut être définie simplement de manière réursive:

Soit un sous-arbre de racine n_i ,

- si la valeur recherchée est celle de la racine n_i , alors la recherche est terminée.

On a trouvé le nœud recherché.

- sinon, si n_i est une feuille (pas de fils) alors la recherche est infructueuse et l'algorithme se termine.

- si la valeur recherchée est plus grande que celle de la racine alors on explore le sous-arbre de droite c'est à dire que l'on remplace n_i par son nœud fils de droite et que l'on relance la procédure de recherche à partir de cette nouvelle racine

- de la même manière, si la valeur recherchée est plus petite que la valeur de n_i , on remplace n_i par son nœud fils de gauche avant de relancer la procédure.

Si l'arbre est équilibré chaque itération divise par 2 le nombre de nœuds candidats.

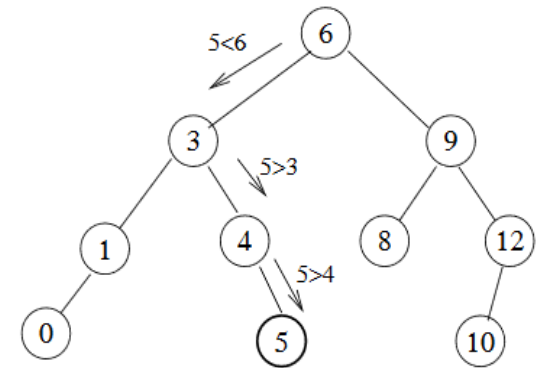
La complexité est donc en $O(\log_2 n)$ si n est le nombre de nœuds de l'arbre

Ajout d'un élément

Pour conserver les propriétés d'un arbre binaire de recherche nécessite de, l'ajout d'un nouvel élément ne peut pas se faire n'importe comment.

L'algorithme récursif d'ajout d'un élément peut s'exprimer ainsi:

- soit x la valeur de l'élément à insérer.
- soit v la valeur du nœud racine n_i d'un sous-arbre.
 - si n_i n'existe pas, le créer avec la valeur x . fin.
 - sinon
 - si x est plus grand que v ,
 - remplacer n_i par son fils droit.
 - recommencer l'algorithme à partir de la nouvelle racine.
 - sinon
 - remplacer n_i par son fils gauche.
 - recommencer l'algorithme à partir de la nouvelle racine



Implémentation

En Langage C, un nœud d'un arbre binaire peut être représenté par une structure contenant champ donnée et deux pointeurs vers les deux nœuds fils

```
struct s_arbre
{
    int valeur;
    struct s_arbre * gauche;
    struct s_arbre * droit;
};
typedef struct s_arbre t_arbre;
```

Implémentation

73

La fonction d'insertion qui permet d'ajouter un élément dans l'arbre et donc de le créer de manière à ce qu'il respecte les propriétés d'un arbre binaire de recherche peut s'écrire ainsi:

```
void insertion(t_arbre ** noeud, int v)
{
    if (*noeud==NULL) /* si le noeud n'existe pas, on le crée */
    {
        *noeud=(t_arbre*) malloc(sizeof(t_arbre));
        (*noeud)->valeur=v;
        (*noeud)->gauche=NULL;
        (*noeud)->droit=NULL;
    }
    else
    {
        if (v>(*noeud)->valeur)
            insertion(&(*noeud)->droit,v ); /* aller a droite */
        else
            insertion(&(*noeud)->gauche,v); /* aller a gauche */
    }
}
```