

# **Techniques avancées de programmation**

Module TB7

Hazem Wannous

(Bureau: D110N)

[hazem.wannous@telecom-lille.fr](mailto:hazem.wannous@telecom-lille.fr)

# Plan

- Partie 1 : Rappel - Types structurés
  - Tableaux, Chaînes de caractères,
  - Structures, Sous-programmes
- Partie 2 : Pointeurs
- Partie 3 : Récursivité
- Partie 4 : Présentation du projet  
“*grapheur d'expressions fonctionnelles*”

# Le langage C

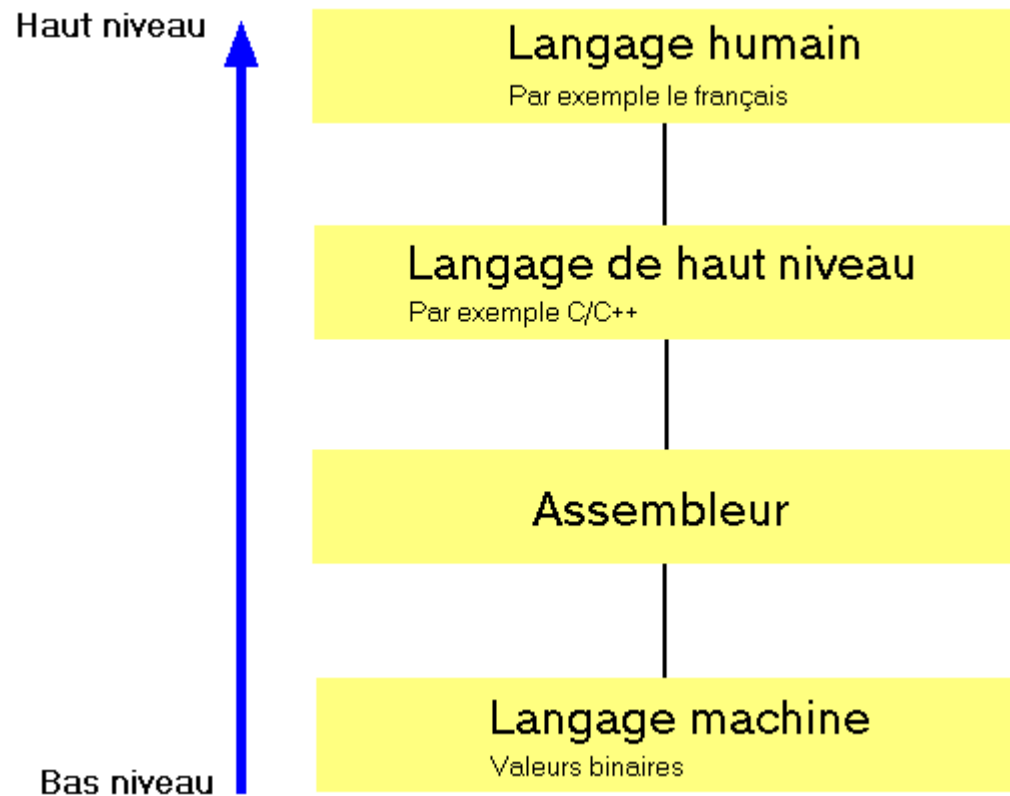
- Langage de bas niveau
  - Accès et manipulation des éléments du processeur (octet, mot, adresse, registre) grâce à un jeu d'opérateurs spécialisés (incrémentation, décrémentation, décalages, manipulation de bits)
  - Accès directe à la mémoire grâce à la notion de pointeur
- Langage « pauvre » en instructions, « riche » en compilateurs et bibliothèques
  - Une quarantaine d'instructions (noyau très restreint) => nombreux compilateurs
  - Pas d'instructions d'E/S, de traitements de chaînes de caractères
  - Pas d'opérateurs pour traiter des vecteurs, matrices et chaînes de caractères (pas d'affectation par exemple)

# Pourquoi programmer en C ?

- Efficacité du code généré
  - Taille du code réduite, temps d'exécution optimal
- Portable
  - Un programme écrit en C est utilisable sur plusieurs processeurs sans modifications (sauf le code spécifique)
  - Compilateurs disponibles pour tous les processeurs
- Utilisé par de nombreux constructeurs, éditeurs de logiciels et la communauté open-source
  - Langage privilégié pour le développement Windows et Unix (API des noyaux spécifiés en C)
- Incontournable dans tous les domaines industriels
  - Contrôle de processus, applications embarquées, traitement du signal, temps réel, ...

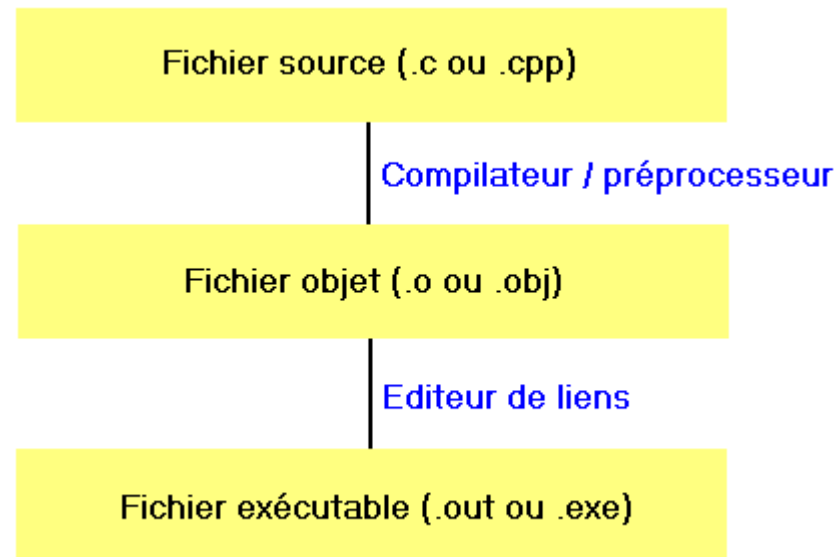
# Compilateurs C/C++

- La "hiérarchie" des niveaux de langages



# Processus de conversion en langage machine

- *Processus de compilation et d'édition de liens*

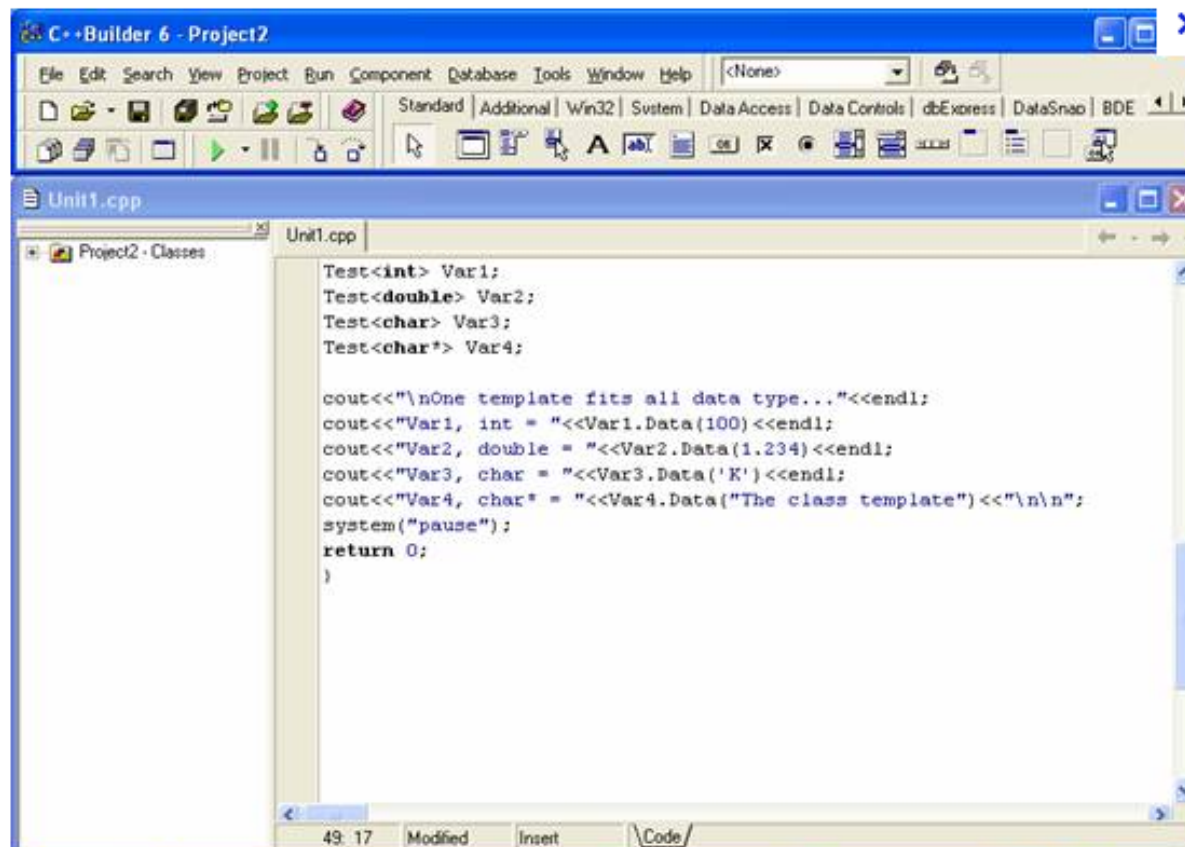


# Compilateurs et programmation sous Linux

- Le compilateur GCC
  - Editeur de texte pour éditer votre code
  - `man gcc`
  - `gcc monprogramme.c -o monexecutable`
  - `./monexecutable`

# Compilateurs et programmation sous Windows

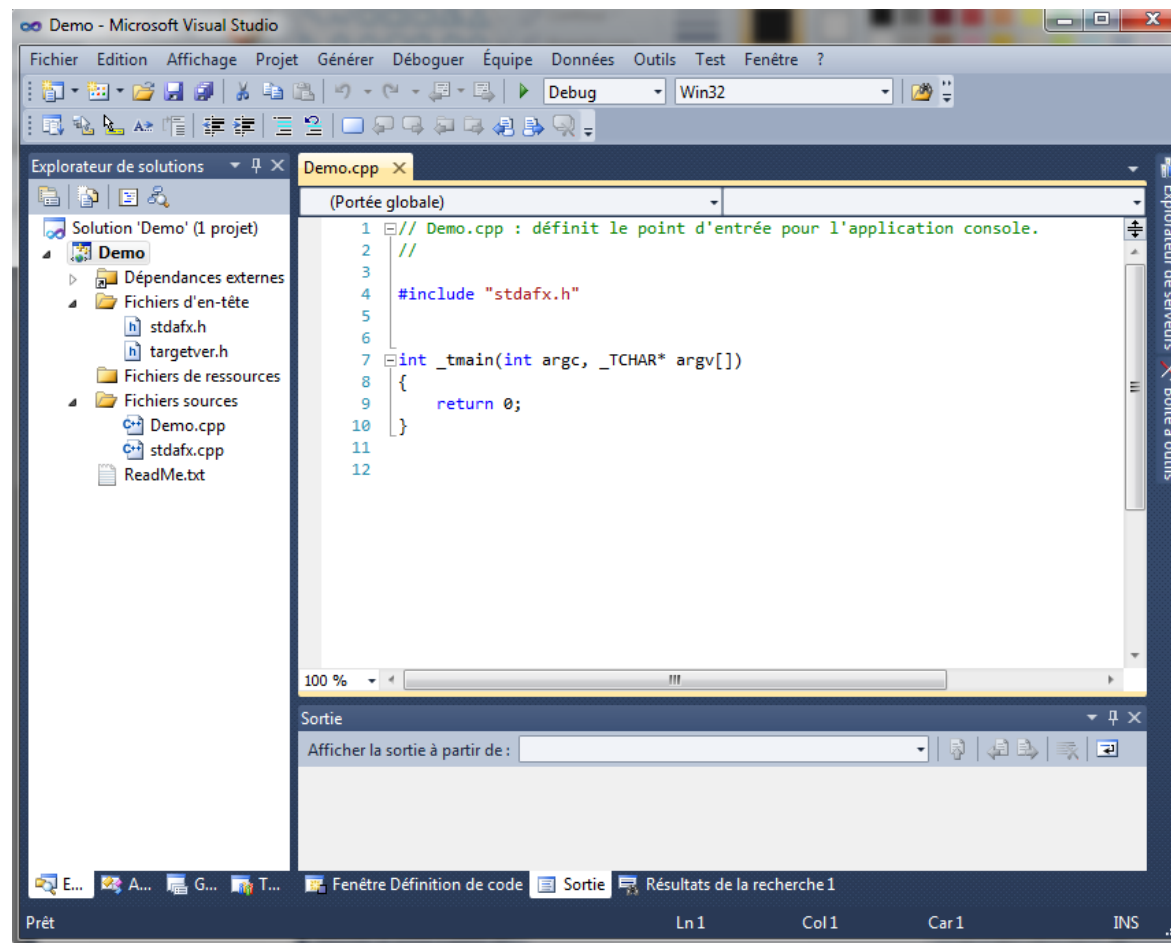
- Borland C++ Builder
  - interface permet de créer le code
  - programme graphique





# Compilateurs et programmation sous Windows

- Microsoft C++
  - interface permet de créer le code
  - programme graphique



# Types structurés

# Le type tableau

- Définition

- Le type tableau est un type de données structurées, c-a-d regroupant plusieurs données sous une même dénomination
- Un tableau est une collection d ' éléments de même type, chaque élément pouvant être repéré par un indice de type entier

- Règles

- La taille du tableau est fixe
- Le type des éléments d' un tableau peut être quelconque
- Un tableau peut avoir de 1 à N dimensions
  - ex. 1 dimension : vecteur
  - ex. 2 dimensions : matrice

# Le type tableau

- Déclaration

- Forme générale : tableau à une dimension
- `<typeElement> <identificateurTableau>[<taille>]`  
    <taille> doit être une valeur connue à la compilation

- Exemple

- `int tabint[10]; /* tableau 10 entiers */`
- `char tabchar[3];`
- `float tabfloat[420]; /* tableau 420 réels */`

- Déclaration avec initialisation

- `int tab1[3] = {1, -5, 6};`
- `float tab2[3] =`  
    `{0.45, 6.78, 789.5}; /*tableau 3 réels */`

# Le type tableau

- Utilisation

- Accès à un élément d' une variable tableau  
    identificateurTableau[indiceElement]
- Premier indice = 0, dernier indice = <taille>-1
- Attention, il n' y aucun contrôle sur l' indice
- Chaque élément d' un tableau peut être employé similairement à une variable de même type
  
- `float v1; int i = 1, j = 10;`
- `v1 = tab2[1] + 3.1415;`
- `scanf("%d", &tab1[4]);`
- `tab1[i + 1] = j * j;`
  
- Il n' y a pas d' opérateurs permettant de manipuler les tableaux dans leur ensemble (pas d' affectation par ex)

# Le type tableau

- Exemple I

```
int main() {  
    int tab[10], indice, pg;  
    printf("Donnez 10 entiers: ");  
    for (indice = 0; indice < 10; indice = indice + 1)  
        scanf("%d", &tab[indice]);  
    pg = tab[0];  
    for (indice = 1; i < 10; indice = indice + 1)  
        if (pg < tab[indice])  
            pg = tab[indice];  
    printf("Le plus grand est: %d ", pg);  
}
```

# Le type tableau

- Exemple 2

```
main() {
    int source[10], copie[10], indice;
    printf("Donnez 10 entiers:");
    for (indice = 0; indice < 10; indice = indice + 1)
        scanf("%d", &source[indice]);
    for (indice = 0; indice < 10; indice = indice + 1)
        copie[indice] = source[indice];

    printf("Affichage de la copie:\n ");
    for (indice = 0; indice < 10; indice = indice + 1)
        printf("%d", copie[indice]);
}
```

# Tableau à N dimensions

- **Tableau à N dimensions**  
<typeElement><identTableau>  
[<taille1>][taille2>]..[<tailleN>  
  
<taillei> fournit la taille de la ième dimension
- **Exemple**  
-float matriceReel[2][3];

matriceReel[0][0]	matriceReel[0][1]	matriceReel[0][2]
matriceReel[1][0]	matriceReel[1][1]	matriceReel[1][2]



# Le type tableau

- `int matriceEntiers[3][2] = {{3,12}, {5, 2}, {11, 4}};`

<code>matriceEntiers[0][0]=3</code>	<code>matriceEntiers[0][1]=12</code>
<code>matriceEntiers[1][0]=5</code>	<code>matriceEntiers[1][1]=2</code>
<code>matriceEntiers[2][0]=11</code>	<code>matriceEntiers[2][1]=4</code>

- Utilisation similaire à celle d'un tableau à 1 dimension
- `matriceEntiers[2][1] = 13;`
- `printf("%d", matriceEntiers[1][0]);`

# Le type tableau

- Exemple

```
main() {  
    /* saisie d'une matrice et multiplication par 10 */  
    int i, j, mat[5][5];  
    for (i = 0; i < 5; i = i + 1) {  
        printf("Entrez la ligne %d :\n", i);  
        for (j = 0; j < 5; j = j + 1)  
            scanf("%d", &mat[i][j]);  
    }  
    for (i = 0; i < 5; i = i + 1) {  
        for (j = 0; j < 5; j = j + 1)  
            mat[i][j] = mat[i][j] * 10;  
    }  
}
```

# Le type tableau

- Chaîne de caractères
  - En C, une chaîne de caractères est représentée par un tableau de caractères et une convention pour indiquer la fin de la chaîne
  - «HELLO»
  - Cette convention est adoptée pour les chaînes de caractères constantes et par les fonctions de la bibliothèque standard

'H'	'E'	'L'	'L'	'O'	'\0'
-----	-----	-----	-----	-----	------

# Le type tableau

- Déclaration

- char tabCar[10];
  - char message[15] = "Hello World";
  - char tabChaine[10][128]; /\* tableau de chaînes \*/

- Utilisation

- Accès aux éléments

- tabCar[0] = 'C';
    - printf("%c", tabCar[7]);

- Ecriture d'une chaîne : code format %s

- printf("Voici la chaine: %s", message);

- Lecture d'une chaîne

- scanf("%s", tabCar); /\* nom du tableau \*/

# Le type tableau

- Principales fonctions de la bibliothèque string.h
  - strlen : retourne la longueur de la chaîne
  - strcpy: copie une chaîne dans une autre
  - strcat : concatène une chaîne à une autre
  - strcmp : compare deux chaînes
  - strchr : retrouve la première occurrence d' un caractère dans une chaîne
  - strstr : retrouve la première occurrence d' une sous-chaîne dans une chaîne

# Le type tableau

- Exemple

```
-#include <string.h>
-#include <stdio.h>
-int main() {
    char chaine1[80], chaine2[80];
    int rescmp;
    printf("Donnez la chaine 1 :"); scanf("%s", chaine1);
    printf("Donnez la chaine 2 :"); scanf("%s", chaine2);
    rescmp = strcmp(chaine1, chaine2);
    if (rescmp == 0)
        printf(" %s = %s ", chaine1, chaine2);
    else if (rescmp > 0)
        printf(" %s > %s ", chaine1, chaine2);
    else
        printf(" %s < %s ", chaine1, chaine2);
-}
```

- Ex. Ecrire dans une chaine:

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

int main(int argc, char *argv[])
{
    char chaine[100];
    int age = 15;

    // On écrit "Tu as 15 ans" dans chaine
    sprintf(chaine, "Tu as %d ans !", age);

    // On affiche chaine pour vérifier qu'elle contient bien cela :
    printf("%s", chaine);

    return 0;
}
```

- Ex. Concaténation:

```
int main(int argc, char *argv[])
{
    /* On crée 2 chaînes. chaine1 doit être assez grande pour accueillir
    le contenu de chaine2 en plus, sinon risque de plantage */
    char chaine1[100] = "Salut ", chaine2[] = "Mateo21";

    strcat(chaine1, chaine2); // On concatène chaine2 dans chaine1

    // Si tout s'est bien passé, chaine1 vaut "Salut Mateo21"
    printf("chaine1 vaut : %s\n", chaine1);
    // chaine2 n'a pas changé :
    printf("chaine2 vaut toujours : %s\n", chaine2);

    return 0;
}
```



- Ex. Concaténation:

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

#include <stdio.h>

int main() {
    char string1[20];
    char string2[20];

    strcpy(string1, "Hello");
    strcpy(string2, "Hellooo");

    printf("Returned String : %s\n", strncat( string1, string2, 4 ));
    printf("Concatenated String : %s\n", string1 );

    return 0;
}
```

- Ex. Copie d'une chaîne dans une autre:

```
int main(int argc, char *argv[])
{
    /* On crée une chaîne "chaîne" qui contient un peu de texte
    et une copie (vide) de taille 100 pour être sûr d'avoir la place
    pour la copie */
    char chaîne[] = "Texte", copie[100] = {0};

    strcpy(copie, chaîne); // On copie "chaîne" dans "copie"

    // Si tout s'est bien passé, la copie devrait être identique à chaîne
    printf("chaîne vaut : %s\n", chaîne);
    printf("copie vaut : %s\n", copie);

    return 0;
}
```

- Ex. Longueur de la chaîne:

```
int main(int argc, char *argv[])
{
    char chaine[] = "Salut";
    int longueurChaine = 0;

    // On récupère la longueur de la chaîne dans longueurChaine
    longueurChaine = strlen(chaine);

    // On affiche la longueur de la chaîne
    printf("La chaîne %s fait %d caractères de long", chaine, longueurChaine);

    return 0;
}
```

# Les énumération

- Définition

- Un type énumération permet de définir des listes de noms symboliques correspondant à des valeurs entières
- Simple définition de quelques constantes symboliques pour des entiers
- Ce type de définitions est une alternative à la définition de constantes avec la directive **#define** du préprocesseur C

- Exemple:

```
enum booleen {faux, true, vrai, false = vrai - 1 };
```

```
enum e1 {  
    bleu = 0, // #define bleu 0  
    blanc = 1, // #define blanc 1  
    rouge = 2 // #define rouge 2  
};
```

# Les unions

- Définition
  - Une union permet de désigner un seul espace mémoire, et de permettre d'y accéder de plusieurs moyens différents. En somme, une union est un regroupement de plusieurs types de données, qui permettent tous d'accéder au même emplacement mémoire
- ```
union
{
    short a;
    long b;
    float c;
} mon_union;
```

# Structures

- Définition

- Une structure est une variable composée de plusieurs champs qui sert à représenter un objet réel ou un concept.

Par exemple une voiture peut être représentée par les renseignements suivants : la marque, la couleur, l'année, . . .

- Une structure regroupe des données de différents types sous un même identificateur

ex1 : Adresse(numero, rue, code\_postal, ville)

ex2 : Personne(nom, prenom, date\_naissance)

- Un type struct permet de définir des modèles de structures

- Forme générale

```
struct <identificateurType> {  
    <déclarations de champs>  
}
```

<identificateurType> est le nom du type

<déclarations de champs> comme des déclarations de variables sans initialisation

# Structures

- Exemple

```
/* définition de types structure */
struct date {int jour; int mois; int annee;};

struct ouvrage {
    int numero;
    char titre[50]; /* tableau dans structure */
    int annee;
};

/* déclaration variable ayant un type structure */
struct date d1, d2;
struct ouvrage o1, o2;
```

# Structures

- Représentation

|    |      |      |       |
|----|------|------|-------|
| d1 | jour | mois | annee |
|    | 12   | 11   | 2002  |

|    |      |      |       |
|----|------|------|-------|
| d2 | jour | mois | annee |
|    | 23   | 2    | 2003  |

|    |        |          |          |     |           |       |
|----|--------|----------|----------|-----|-----------|-------|
| o1 | numero | titre[0] | titre[1] | ... | titre[49] | annee |
|    | 1234   | 'S'      | 'E'      | ... | '\0'      | 1970  |



# Structures

- Utilisation
  - Déclaration avec initialisation
  - `struct` ouvrage o1={1023,"Le langage C", 1983};
- Affectation entre structures (contrairement aux tableaux)
- Attention !
  - Pas d'opérateur d'égalité (==), d'inégalité (!=), ni de relation d'ordre entre structures
  - Pas de lecture/écriture associé
  - => A programmer manuellement

# Structures

- Affectation entre structures / entre tableaux

```
struct data
{
    short a;
    float y;
    long d;
};
```

```
int main() {
```

```
    struct data a, b;
```

```
        a.a = 10;
        a.y = 3.14;
        a.d = 567;
        ...
        b = a; /* ok */
```

```
}
```

```
int tab[5];
int tab2[5]={1,2,3,5,8}; /*
ok */
tab = tab2; /* incorrect */
```

# Structures

- Accès
  - L' accès aux champs d' une variable structure s' effectue en faisant suivre l' identificateur de variable d' un « . » et du nom du champ à accéder
  - Chaque champ d' une structure peut être employé similairement à une variable du même type
  - Exemple

```
d1.jour = 10;
printf(«%s », o1.titre);
scanf(«%d », &o1.numero);
if (d1.annee < d2.annee) ...
```
  - Forme générale

```
identificateurStructure.champ
```

# Structures

- Exemple

```
#include<stdio.h>

struct date { int jour; int mois; int annee; };

int main() {
    struct date d1, d2;
    int egal;
    scanf(" %d %d %d", &d1.jour, &d1.mois, &d1.annee);
    scanf(" %d %d %d", &d2.jour, &d2.mois, &d2.annee);
    if ((d1.annee == d2.annee) &&
        (d1.mois == d2.mois) && (d1.jour == d2.jour))
        egal = 1;
    else
        egal = 0;
    if (egal) printf("les deux dates sont egales");
}
```

# Tableaux de structures

- **Déclaration**  
-`struct` ouvrage bibliotheque[10];
- **Accès aux éléments du tableau**  
`int` i = 1; `struct` ouvrage o1;  
bibliotheque[i + 10] = o1;  
o1 = bibliotheque[i];
- **Accès aux champs des éléments du tableau**  
-bibliotheque[3].numero = 1024;  
-printf("%s", bibliotheque[3].titre);  
-scanf("%d", &bibliotheque[1].annee);

# Structures

- Example

```
#include<stdio.h>
enum videoType {Film, Documentaire, Concert};
struct video {
    int numero;
    char titre[80];
    int annee;
    videoType genre;
};
struct video videotheque[10];
int i; char g;

main() {
    for(i = 0; i < 10; i = i + 1) {
        printf("Saisie k7 num: %d", i);
        printf("Numero:"); scanf("%d",&(videotheque[i].numero));
        printf("Titre:"); scanf("%s", videotheque[i].titre);
        printf("Genre[F]ilm,[D]ocument,[C]oncert:"); scanf("%c", &g);
        switch(g) {
            case 'F':videotheque[i].genre=Film;break;
            case 'G':videotheque[i].genre=Documentaire;break;
            case 'C':videotheque[i].genre=Concert;break;
        }
    }
}
```

# Structures

- Imbrication des structures

```
struct personne {  
    char nom[20];  
    struct date dateNaissance;  
};  
struct personne pierre, paul;
```

- Accès aux champs de la structure imbriquée

```
-pierre.dateNaissance.mois = 1;  
-printf("%d", pierre.dateNaissance.jour);  
-scanf("%d", &pierre.dateNaissance.annee);  
-paul.dateNaissance = d1;
```

# Nommage de types en C

- On peut donner des noms (synonymes) à des définition de types par typedef
- **Forme générale**
  - typedef <declaration>
  - <declaration> : même syntaxe qu'une déclaration de variable mais l'identificateur désigne le nom du type
- **Exemple**
  - typedef int entier;
  - typedef int Matrice[10][20];
  - typedef struct date Date;
- **Conséquence au niveau des déclarations (équivalence)**
  - entier i, j; <=> int i, j;
  - Matrice m; <=> int m[10][20];
  - Date d; (sans struct !) <=> struct date d;



- Exemple:

```
#include<stdio.h>
typedef struct
{
    int x;
    int y;
} Coordonnees;

struct MaStructure
{
    Coordonnees element;
    int monBooleen;
    char maChaine[10];
};

int main (void)
{
    struct MaStructure test;

    //
    test.element.x = 6;
    test.element.y = 5;

    return 0;
}
```

# Les fichiers en-tête en C

- Extrait de `stdio.h`

```
/
*****
*****/
/* FORMATTED INPUT/OUTPUT FUNCTIONS */
/
*****
*****/
extern int    fprintf(FILE *_fp, const char *_format, ...);
extern int    fscanf(FILE *_fp, const char *_fmt, ...);
extern int    printf(const char *_format, ...);
extern int    scanf(const char *_fmt, ...);
extern int    sprintf(char *_string, const char *_format, ...);
extern int    sscanf(const char *_str, const char *_fmt, ...);
extern int    vfprintf(FILE *_fp, const char *_format, char *_ap);
extern int    vprintf(const char *_format, char *_ap);
extern int    vsprintf(char *_string, const char *_format, char
* ap);
```

# Les fonctions en C

- Bien sûr, nous pouvons écrire nos propres fonctions.

```
/* Routine de calcul du maximum */
```

```
int imax(int n, int m)
```

```
{
```

```
    int max;
```

```
    if (n>m)
```

```
        max = n;
```

```
    else
```

```
        max = m;
```

```
    return max;
```

```
}
```

**Déclaration de la fonction**

**Variable locale**

**Valeur retournée par la fonction**

# Les fonctions en C

- Fonctions sans arguments et ne retournant pas de valeur.

`void fonction(void)`

- Fonctions avec arguments ne retournant pas de valeur.

`void fonction(int x, int y, char ch)`

- Les fonctions exigent la déclaration d'un prototype avant son utilisation:

```
/* Programme principal */
```

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

```
int imax(int,int);
```

```
main()
```

```
{ ... }
```

```
int imax(int n, int m)
```

```
{ ... }
```

**Prototype de la fonction**

**La fonction est définie ici**

# Boucle « for »

```
/* Boucle for */  
#include <stdio.h>  
#define NUMBER 22  
main()  
{  
    int count, total = 0;  
  
    for(count = 1; count <= NUMBER; count++, total += count)  
        printf("Vive le langage C !!!\n");  
    printf("Le total est %d\n", total);  
}
```

Initialisation

Condition de fin  
de boucle

Incrémentation et autres fonctions

# Boucle « while »

```
/* Boucle while */
#include <stdio.h>
#define NUMBER 22
main()
{
    int count = 1, total = 0;

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

Initialisation

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

Incrémentation

# Boucle « do while »

```
/* Boucle do while */
#include <stdio.h>
#define NUMBER 22
main()
{
    int count = 1, total = 0;

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

**Initialisation**

**Incrémentation**

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

# Choix multiple: « switch case »

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

```
main()
```

```
{
```

```
  char choix;
```

```
  ...
```

```
  switch(choix)
```

```
  {
```

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

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

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

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

```
  }
```

```
}
```

Paramètre de décision

Exécuté si choix = a

Exécuté si choix = a ou b

Exécuté si choix = a, b ou c

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



# Effet du « break »

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

```
main()
```

```
{
```

```
  char choix;
```

```
  ...
```

```
  switch(choix)
```

```
  {
```

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

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

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

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

```
  }
```

```
}
```

Paramètre de décision

Exécuté si choix = a

Exécuté si choix = b

Exécuté si choix = c

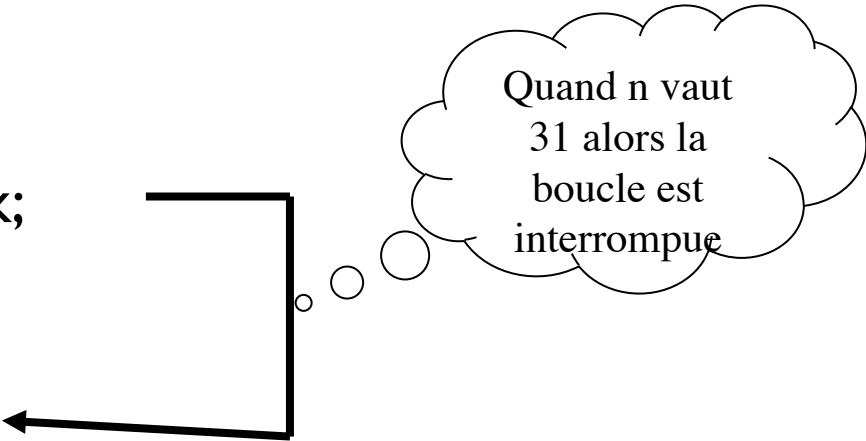
Exécuté si choix non  
répertorié par un « case »

# Effet du « break »

- l'instruction break
  - permet d'interrompre prématurément une boucle et de se brancher vers la première instruction n'appartenant pas à la boucle

- exemple:

```
int i;int n=20;  
for (i=0;i<10;i++)  
{  
    if (n==31) break;  
    n=n+2;  
}  
cout <<n<<"\n";
```



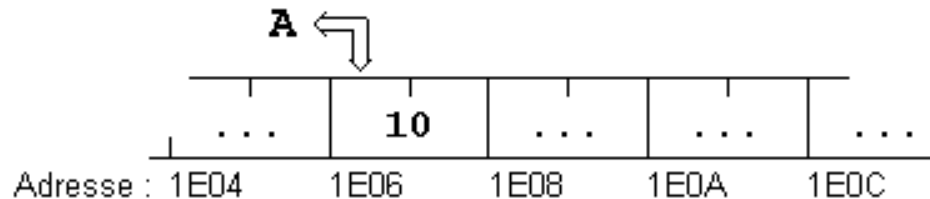
Quand n vaut  
31 alors la  
boucle est  
interrompue

# ***Les pointeur***

# L'adressage

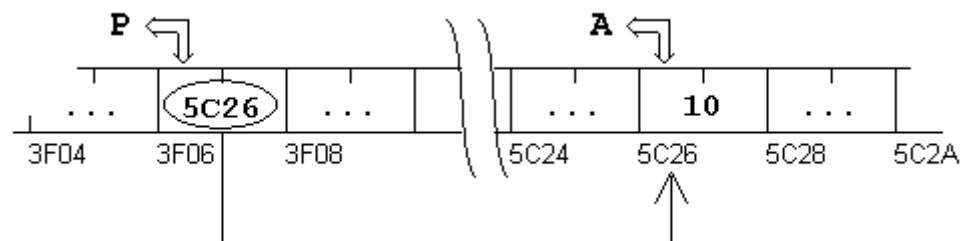
- **Adressage direct :**
  - Accès au contenu d'une variable par le nom de la variable
  - Exemple:

Short A;  
A = 10;



- **Adressage indirect :**
  - Accès au contenu d'une variable, en passant par un pointeur qui contient l'adresse de la variable
  - Exemple:

Short A;  
A = 10;  
P = &A;



# Le pointeurs

## □ *Un **pointeur***

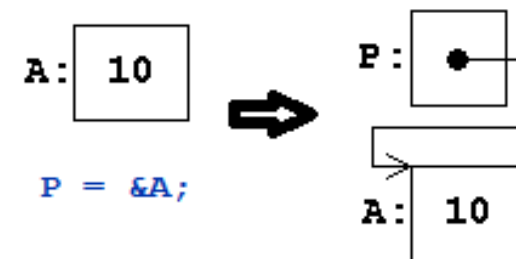
- *Variable spéciale qui peut contenir l'adresse d'une autre variable*
- chaque pointeur est limité à un type de données
- Si un pointeur P contient l'adresse d'une variable A, on dit que '*P pointe sur A*'.
- Les pointeurs et les noms de variables ont le même rôle: (Ils donnent accès à un emplacement dans la mémoire interne de PC)
  - ✓ *Le **pointeur*** : variable qui peut 'pointer' sur différentes adresses.
  - ✓ *Nom d'une variable* : toujours lié à la même adresse

# Les opérateurs de base

## □ **L'opérateur 'adresse de' : &**

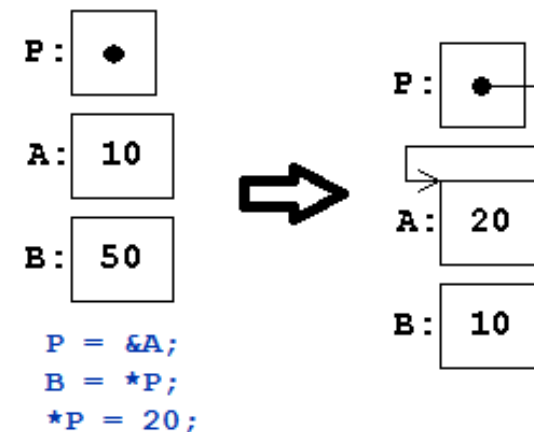
- <NomVariable> fournit l'adresse de la variable <NomVariable>
- Exemple

```
int N;  
printf("Entrez un nombre entier : ");  
scanf("%d", &N);
```



## □ **L'opérateur 'contenu de' : \***

- \*<NomPointeur> désigne le contenu de l'adresse référencée par le pointeur <NomPointeur>



# Le pointeurs

## ❑ **Déclaration d'un pointeur**

- `<Type> *<NomPointeur>` déclare un pointeur `<NomPointeur>` qui peut recevoir des adresses de variables du type `<Type>`
- Chaque pointeur est limité à un type de données

- Exemple

`int *PNUM;` ♪

"\*PNUM est du type int"

ou


"PNUM est un pointeur sur int"

ou

"PNUM peut contenir l'adresse d'une variable du type int"

## ❑ **Priorité de \* et &**

- Même priorité que les autres opérateurs unaires
- Exemple

|                          |                                                                                       |                         |                   |                       |
|--------------------------|---------------------------------------------------------------------------------------|-------------------------|-------------------|-----------------------|
| <code>P = &amp;X;</code> |  | <code>Y = *P+1</code>   | $\Leftrightarrow$ | <code>Y = X+1</code>  |
|                          |                                                                                       | <code>*P = *P+10</code> | $\Leftrightarrow$ | <code>X = X+10</code> |
|                          |                                                                                       | <code>*P += 2</code>    | $\Leftrightarrow$ | <code>X += 2</code>   |
|                          |                                                                                       | <code>++*P</code>       | $\Leftrightarrow$ | <code>++X</code>      |
|                          |                                                                                       | <code>(*P)++</code>     | $\Leftrightarrow$ | <code>X++</code>      |

# Le pointeurs

## □ *Adressage d'un tableau*

- Le nom d'un tableau représente l'adresse de son premier élément  
&tableau[0] et tableau sont une seule et même adresse
- Exemple:  
**int A[10]; int \*P;** → **P = A;** est équivalente à **P = &A[0];**
  - **\*(P+1)** désigne le contenu de **A[1]**
  - **\*(P+2)** désigne le contenu de **A[2]**
  - ... ..
  - **\*(P+i)** désigne le contenu de **A[i]**



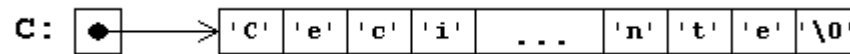
# Le pointeurs

## □ **Pointeurs et chaînes de caractères**

### ▪ **Affectation / initialisation:**

char \*C; C = "Ceci est une chaîne de caractères constante";

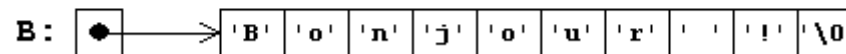
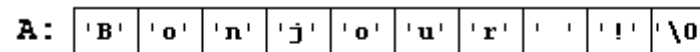
ou char \*C = "Ceci est une chaîne de caractères constante";



### ▪ **Différence entre:**

char A[ ] = "Bonjour !"; /\* A est un tableau \*/

char \*B = "Bonjour !"; /\* B est un pointeur \*/



### ▪ **Modification**

1 char \*A = "Petite chaîne";  
char \*B = "chaîne plus longue"; )→ A = B ?

2 char A[45] = "Petite chaîne";  
char B[45] = "Deuxième chaîne un peu plus longue";  
char C[30];  
A = B; /\* IMPOSSIBLE -> ERREUR !!! \*/  
C = "Bonjour !"; /\* IMPOSSIBLE -> ERREUR !!! \*/

# Récurtivité

# Définition de la récursivité

- Algorithme qui comporte
  - au moins un appel à lui même
  - au moins une condition d'arrêt
- Exemple : factorielle
  - $n! = n * (n-1) !$
  - $0! = 1$

# Écriture d'un algorithme récursif

```
long factorielle(int n){  
    if (n<=0) return 1;  
    else return n*factorielle(n-1);}
```

CONDITION D'ARRET

APPEL RECURSIF

factorielle(5) = 5\*factorielle(4)                      5\*24

factorielle(4) = 4\*factorielle(3)                      4\*6

factorielle(3) = 3\*factorielle(2)                      3\*2

factorielle(2) = 2\*factorielle(1)                      2\*1

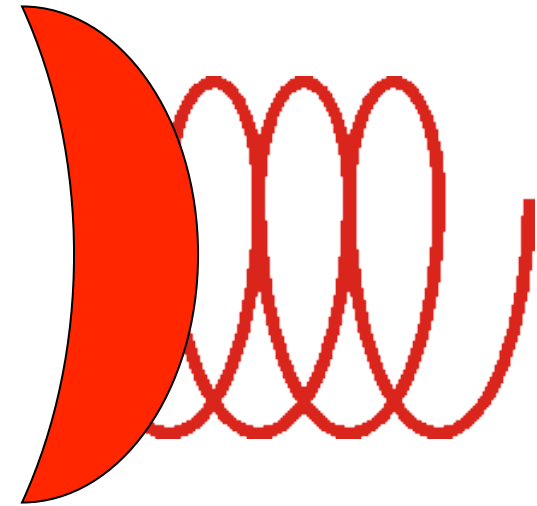
factorielle(1) = 1\*factorielle(0)                      1\*1

# Algorithmes récurrents

## la mise en œuvre informatique

```
long factorial(int n){  
    if (n<=0) return 1;  
    return n*factorial(n-1);  
}
```

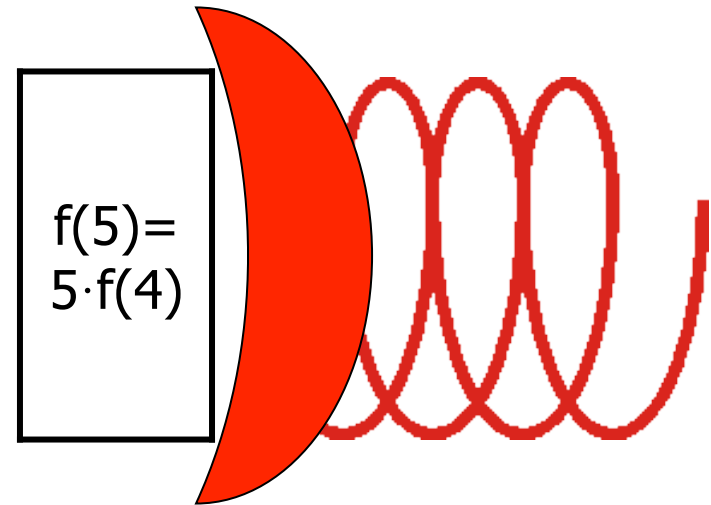
Compute 5!



# Algorithmes récurrents

## la mise en œuvre informatique

```
long factorial(int n){  
    if (n<=0) return 1;  
    return n*factorial(n-1);  
}
```

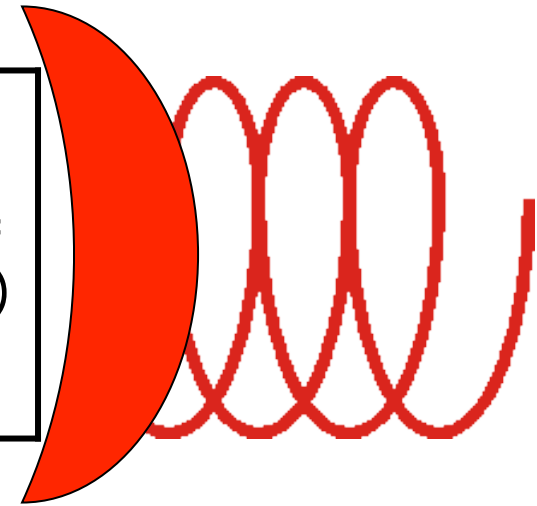


# Algorithmes récurrents

## la mise en œuvre informatique

```
long factorial(int n){  
    if (n<=0) return 1;  
    return n*factorial(n-1);  
}
```

|                       |                       |
|-----------------------|-----------------------|
| $f(4) = 4 \cdot f(3)$ | $f(5) = 5 \cdot f(4)$ |
|-----------------------|-----------------------|

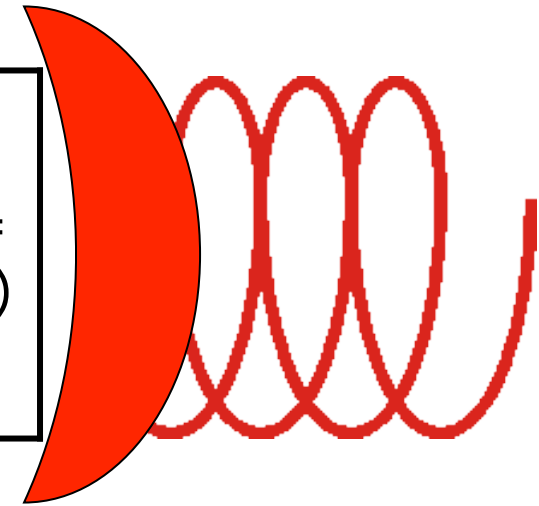


# Algorithmes récurrents

## la mise en œuvre informatique

```
long factorial(int n){  
    if (n<=0) return 1;  
    return n*factorial(n-1);  
}
```

|                           |                           |                           |
|---------------------------|---------------------------|---------------------------|
| $f(3)=$<br>$3 \cdot f(2)$ | $f(4)=$<br>$4 \cdot f(3)$ | $f(5)=$<br>$5 \cdot f(4)$ |
|---------------------------|---------------------------|---------------------------|



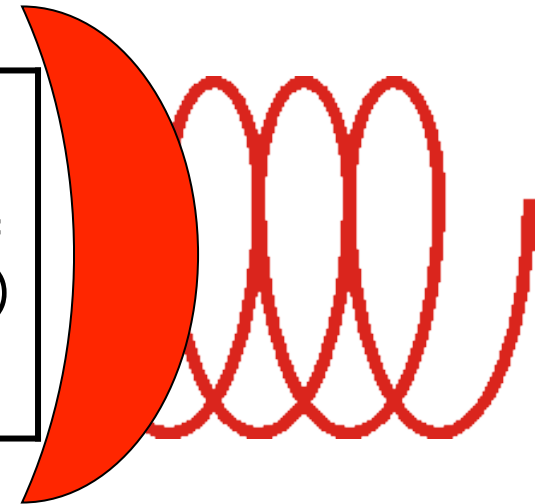


# Algorithmes récurrents

## la mise en œuvre informatique

```
long factorial(int n){  
    if (n<=0) return 1;  
    return n*factorial(n-1);  
}
```

|                           |                           |                           |                           |
|---------------------------|---------------------------|---------------------------|---------------------------|
| $f(2)=$<br>$2 \cdot f(1)$ | $f(3)=$<br>$3 \cdot f(2)$ | $f(4)=$<br>$4 \cdot f(3)$ | $f(5)=$<br>$5 \cdot f(4)$ |
|---------------------------|---------------------------|---------------------------|---------------------------|

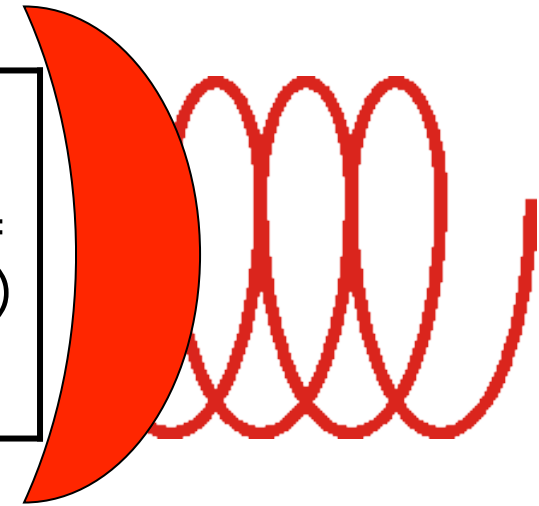


# Algorithmes récurrents

## la mise en œuvre informatique

```
long factorial(int n){  
    if (n<=0) return 1;  
    return n*factorial(n-1);  
}
```

|                           |                           |                           |                           |                           |
|---------------------------|---------------------------|---------------------------|---------------------------|---------------------------|
| $f(1)=$<br>$1 \cdot f(0)$ | $f(2)=$<br>$2 \cdot f(1)$ | $f(3)=$<br>$3 \cdot f(2)$ | $f(4)=$<br>$4 \cdot f(3)$ | $f(5)=$<br>$5 \cdot f(4)$ |
|---------------------------|---------------------------|---------------------------|---------------------------|---------------------------|

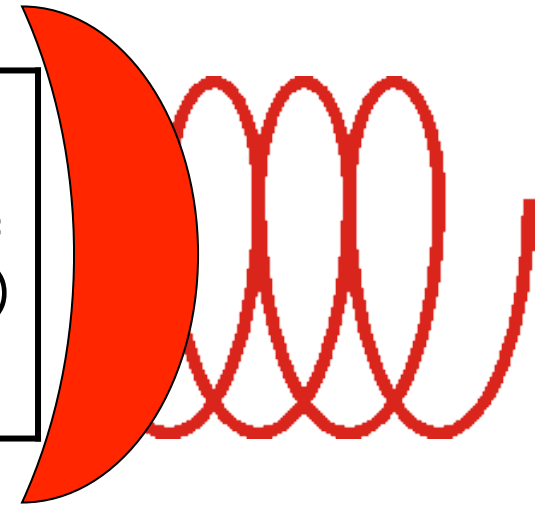


# Algorithmes récurrents

## la mise en œuvre informatique

```
long factorial(int n){  
    if (n<=0) return 1;  
    return n*factorial(n-1);  
}
```

|                            |                           |                           |                           |                           |                           |
|----------------------------|---------------------------|---------------------------|---------------------------|---------------------------|---------------------------|
| $f(0)=$<br>$1 \rightarrow$ | $f(1)=$<br>$1 \cdot f(0)$ | $f(2)=$<br>$2 \cdot f(1)$ | $f(3)=$<br>$3 \cdot f(2)$ | $f(4)=$<br>$4 \cdot f(3)$ | $f(5)=$<br>$5 \cdot f(4)$ |
|----------------------------|---------------------------|---------------------------|---------------------------|---------------------------|---------------------------|

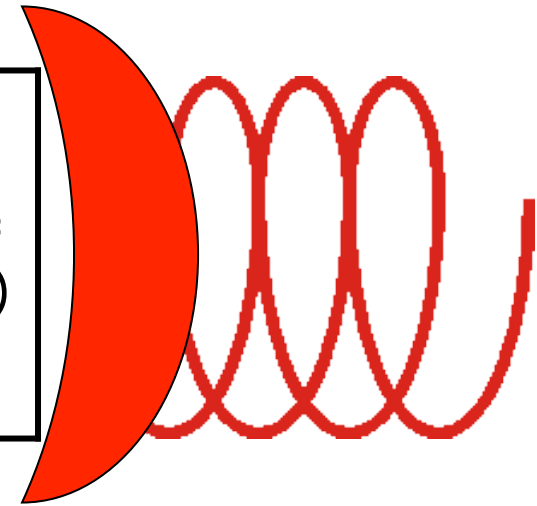


# Algorithmes récurifs

## la mise en œuvre informatique

```
long factorial(int n){  
    if (n<=0) return 1;  
    return n*factorial(n-1);  
}
```

|                                  |                            |                            |                            |                            |
|----------------------------------|----------------------------|----------------------------|----------------------------|----------------------------|
| $1 \cdot 1 =$<br>$1 \rightarrow$ | $f(2) =$<br>$2 \cdot f(1)$ | $f(3) =$<br>$3 \cdot f(2)$ | $f(4) =$<br>$4 \cdot f(3)$ | $f(5) =$<br>$5 \cdot f(4)$ |
|----------------------------------|----------------------------|----------------------------|----------------------------|----------------------------|

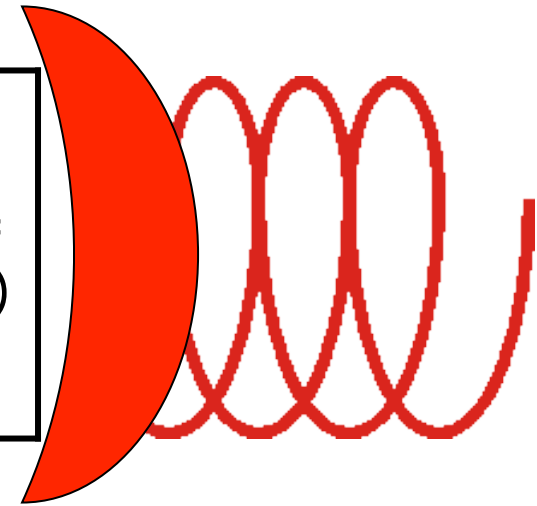


# Algorithmes récurrents

## la mise en œuvre informatique

```
long factorial(int n){  
    if (n<=0) return 1;  
    return n*factorial(n-1);  
}
```

|                                  |                            |                            |                            |
|----------------------------------|----------------------------|----------------------------|----------------------------|
| $2 \cdot 1 =$<br>$2 \rightarrow$ | $f(3) =$<br>$3 \cdot f(2)$ | $f(4) =$<br>$4 \cdot f(3)$ | $f(5) =$<br>$5 \cdot f(4)$ |
|----------------------------------|----------------------------|----------------------------|----------------------------|

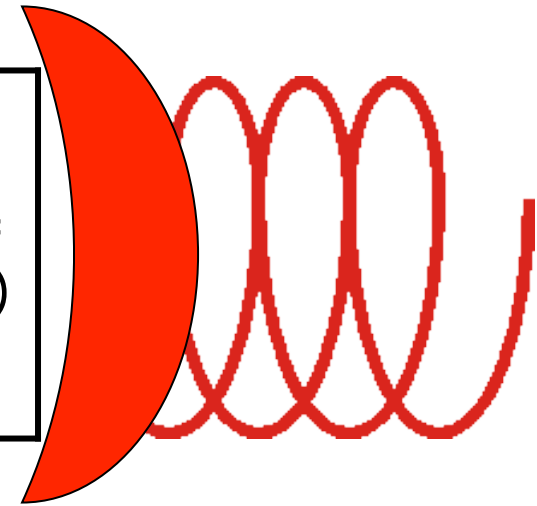


# Algorithmes récurrents

## la mise en œuvre informatique

```
long factorial(int n){  
    if (n<=0) return 1;  
    return n*factorial(n-1);  
}
```

|                             |                       |                       |
|-----------------------------|-----------------------|-----------------------|
| $3 \cdot 2 = 6 \rightarrow$ | $f(4) = 4 \cdot f(3)$ | $f(5) = 5 \cdot f(4)$ |
|-----------------------------|-----------------------|-----------------------|

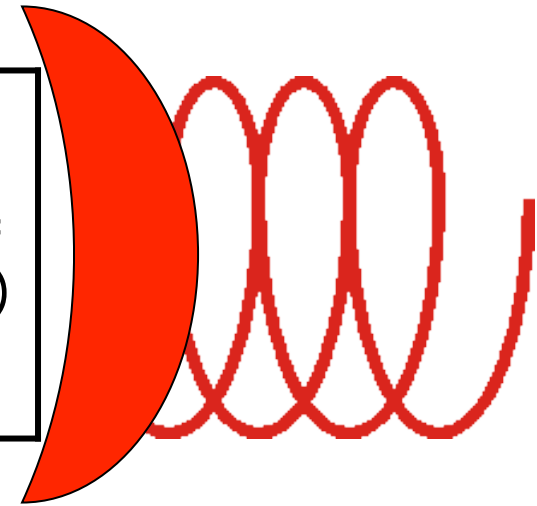


# Algorithmes récurrents

## la mise en œuvre informatique

```
long factorial(int n){  
    if (n<=0) return 1;  
    return n*factorial(n-1);  
}
```

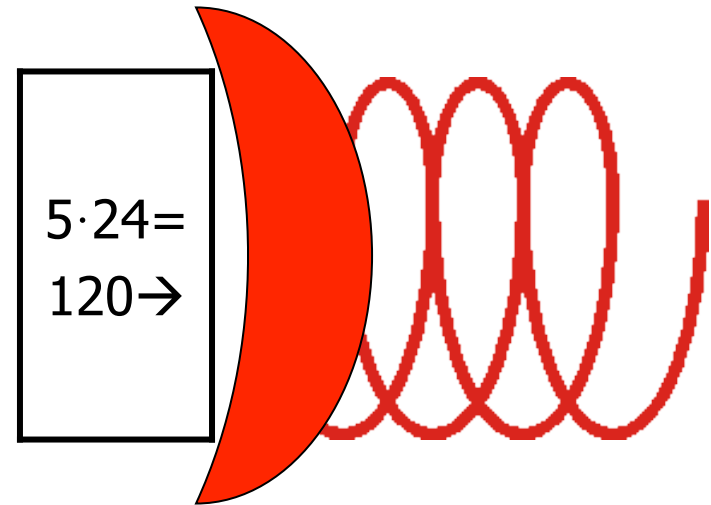
|             |                 |
|-------------|-----------------|
| 4·6=<br>24→ | f(5)=<br>5·f(4) |
|-------------|-----------------|



# Algorithmes récurrents

## la mise en œuvre informatique

```
long factorial(int n){  
    if (n<=0) return 1;  
    return n*factorial(n-1);  
}
```

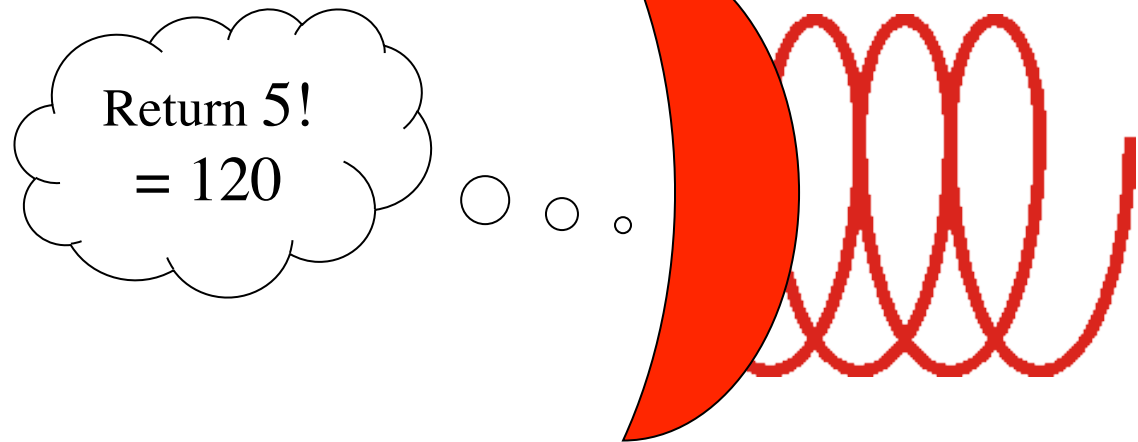




# Algorithmes récurrents

## la mise en œuvre informatique

```
long factorial(int n){  
    if (n<=0) return 1;  
    return n*factorial(n-1);  
}
```



# Un autre exemple

**Niveau 1**

**Niveau 2**

**Niveau 3**

**Niveau 4**

**NIVEAU 4**

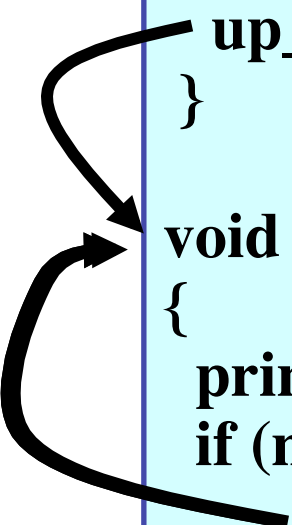
**NIVEAU 3**

**NIVEAU 2**

**NIVEAU 1**

```
/* Programme principal */
#include <stdio.h>
void up_and_down(int);
main()
{
    up_and_down(1)
}

void up_and_down(int n)
{
    printf("Niveau %d\n" n);
    if (n<4)
        up_and_down(n+1);
    printf("NIVEAU %d\n" n);
}
```



# Empilage du contexte local

- Chaque procédure stocke dans une zone mémoire
  - les paramètres
  - les variables locales
- Cette zone s'appelle la pile car les données sont
  - empilées lors de l'appel d'une procédure
  - désempilées à la fin de la procédure
- A chaque appel récursif, l'ordinateur empile donc
  - les variables locales
  - les paramètres qui doivent changer à chaque appel

# Transformer une boucle en une procédure récursive

- Procédure itérative :

```
void compter() {  
    for (i=1;i<10;i++)  
        cout << i ;}
```

- Procédure récursive équivalente

```
void compter_rec(int i) {  
    cout << i;  
    if(i<10)  
        compter_rec(i+1) ;}
```

# Inverser une chaîne de caractères

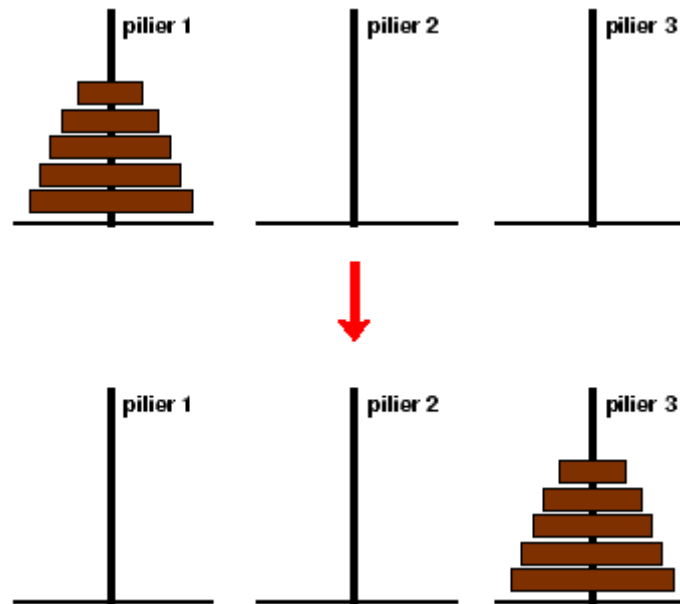
- ```
string inverse(string str) {  
    int l= strlen(str);  
    if (l<=1) return str;  
    else return inverse(str.substr(1,l))+ str[0];}
```

CONDITION D' ARRET

APPEL RECURSIF

- ```
inverse("abcd")-> inverse("bcd")+ "a"  
    inverse("bcd")-> inverse("cd")+ "b"  
        inverse("cd")-> inverse("d")+ "c"  
            dc  
        dcb  
    dcba
```

# Tours de Hanoi (I)

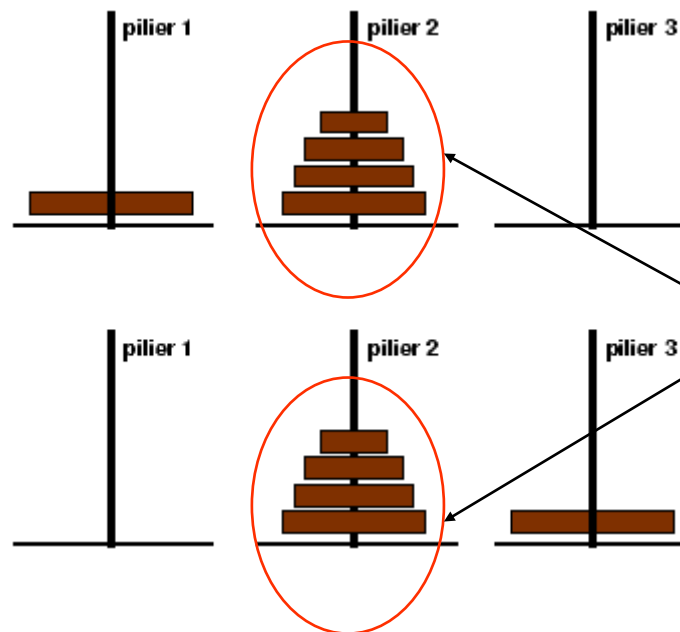


- Déplacer les disques d'un pilier à un autre
- Un disque d'un certain diamètre ne peut pas être placé au dessus d'un disque de diamètre inférieur.

## Tours de Hanoi (2)

Si on a "n" disques à déplacer :

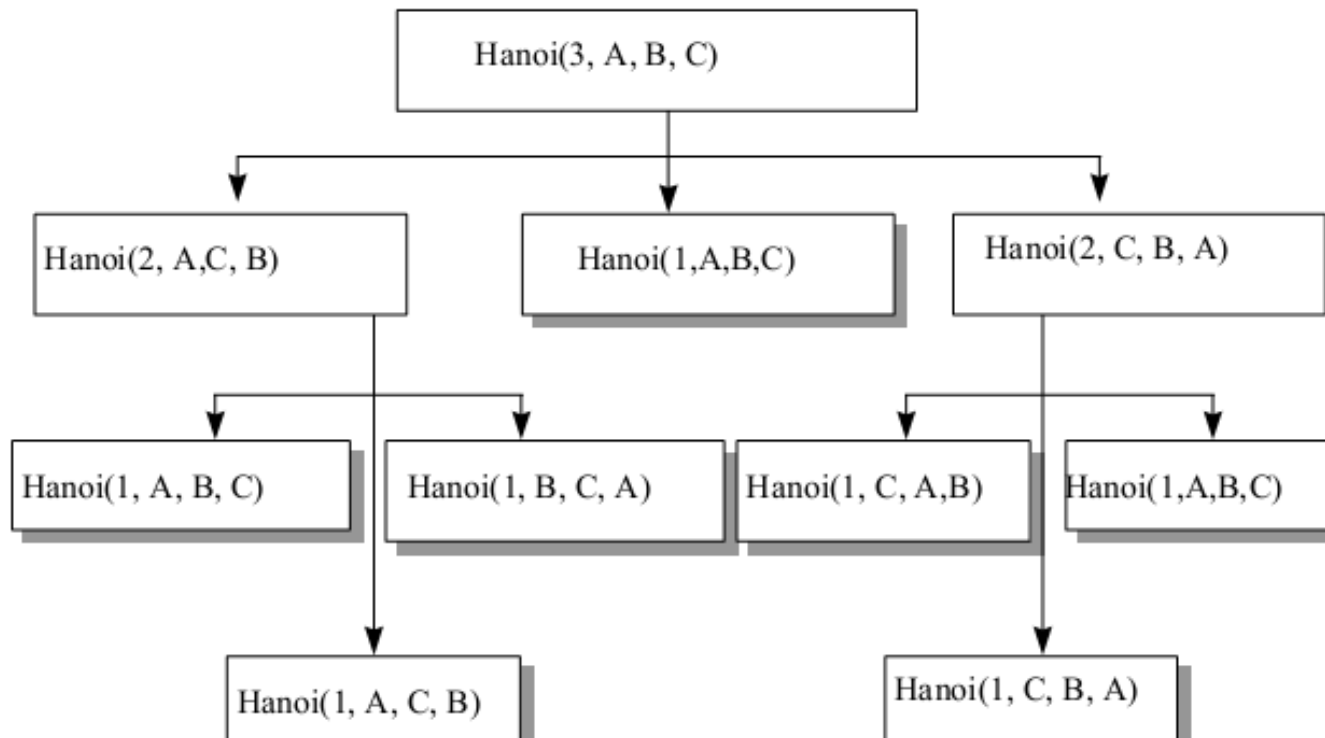
- on déplace "n-1" disques vers le pilier intermédiaire
- on déplace le disque "n" du pilier initial vers le pilier final
- on déplace les "n-1" disques du pilier intermédiaire vers le pilier final



Hanoi avec  
un disque de moins

# Tours de Hanoi (3)

Pseudo-code qui illustre les différents appels





# Tours de Hanoï (3)

Code C:

```
void Hanoi(int NbrePlateau, char Src, char Dst, char Tmp)
{
    if ( NbrePlateau==1)
        printf("deplacer le plateau de %c vers %c \n",Src,Dst);
    else
    {
        Hanoi (NbrePlateau-1,Src,Tmp,Dst);
        Hanoi (1,Src,Dst,Tmp);
        Hanoi (NbrePlateau-1,Tmp,Dst,Src);
    }
}
```

# Efficacité de la récursivité ?

- La récursivité est légèrement moins rapide qu'un algorithme itératif équivalent
  - Temps nécessaire à l'empilage et au désempilage des données
- La récursivité utilise plus de ressources mémoire
  - pour empiler les contextes

Mais ...

- La récursivité est plus « élégante »
- Les algorithmes récursifs sont souvent plus faciles à écrire

# **Projet**

**Réalisation d'un *grapheur*  
d'*expressions fonctionnelles***

# Réalisation d'un *grapheur d'expressions fonctionnelles*

□ Logiciel de représentation graphique des expressions

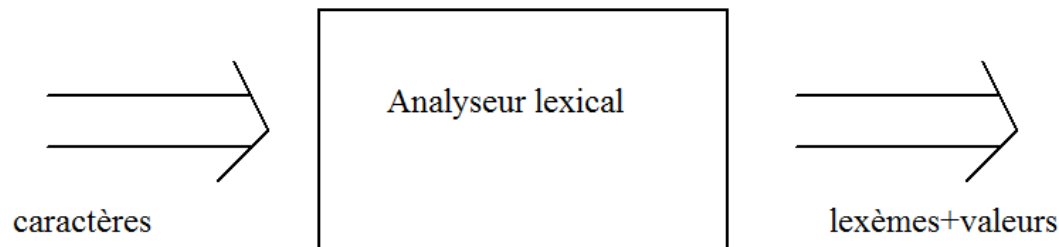
▪ ***Découpage en 4 unités***

- Analyse lexicale
- Analyse syntaxique
- interprétation du code généré
- interface graphique

# Réalisation d'un *grapheur d'expressions fonctionnelles*

## □ Analyse lexicale

- **Entrée** : flux de caractère
- **Sortie** : flux d'entités lexicales (lexèmes ou jetons) (+ éventuellement d'une indication de valeur)
- **Lexèmes**: (REEL, OPERATEUR, FONCTION, ERREUR, FIN, PAR\_OUVR, PAR\_FERM, VARIABLE, NO\_TOKEN, ...)

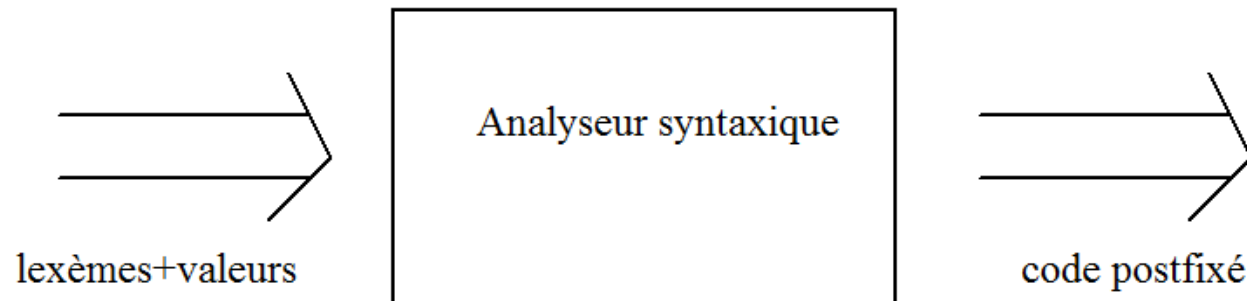


- Elle transforme le texte en une suite de lexèmes (token).

# Réalisation d'un *grapheur d'expressions fonctionnelles*

## □ Analyse syntaxique

- **Entrée** : flux de couples (lexèmes + valeurs)
- **Sortie** : flux de code postfixé (ou arbre binaire)



- Son rôle est de vérifier la conformité de l'expression avec la grammaire définie et de produire en sortie un flux de code postfixé

# Analyse syntaxique

- Grammaire

Règle 1 :  $\text{exp} ::= \text{nombre réel}$

Règle 2 :  $\text{exp} ::= \text{variable}$

Règle 3 :  $\text{exp} ::= \text{fonction } \text{exp}$

Règle 4 :  $\text{exp} ::= ( \text{exp } \text{opérateur } \text{exp} )$

Règle 5 :  $\text{exp} ::= ( \text{exp} )$

Règle 6 :  $\text{expression\_complète} ::= \text{exp fin}$

- Règles de production

Règle 1 : nombre réel

Règle 2 : variable

Règle 3 :  $\text{prod}(\text{exp}) \text{ fonction}$

Règle 4 :  $\text{prod}(\text{exp}) \text{ prod}(\text{exp}) \text{ opérateur}$

Règle 5 :  $\text{prod}(\text{exp})$

Règle 6 :  $\text{prod}(\text{exp})$

# Analyse lexicale et syntaxique

## Jeton.h

```
//énumération des différents types de lexems existants
typedef enum
{
    REEL, OPERATEUR, FONCTION, ERREUR, FIN, PAR_OUV, PAR_FERM, VARIABLE, BAR_OUV, BAR_FERM, ABSOLU
} typelexem;

//énumération des diff types d'opérateurs existants
typedef enum
{
    PLUS, MOINS, FOIS, DIV, PUIS
} typeopérateur;

//énumération des diff types de fonctions existantes
typedef enum
{
    ABS, SIN, SQRT, LOG, COS, TAN, EXP, ENTIER, VAL_NEG, SINC
} typefonction;

//énumération des diff types de valeurs existantes
typedef union
{
    float reel;
    typefonction fonction;
    typeopérateur opérateur;
    typeerreur erreur;
} typevaleur;

//énumération des diff types de jetons existants
typedef struct
{
    typelexem lexem;
    typevaleur valeur;
} typejeton;

//déclaration de l'arbre
typedef struct Node
{
    typejeton jeton;
    struct Node *pjeton_preced;
    struct Node *pjeton_suiv;
} Node;

typedef Node *Arbre;
```



# Analyse syntaxique

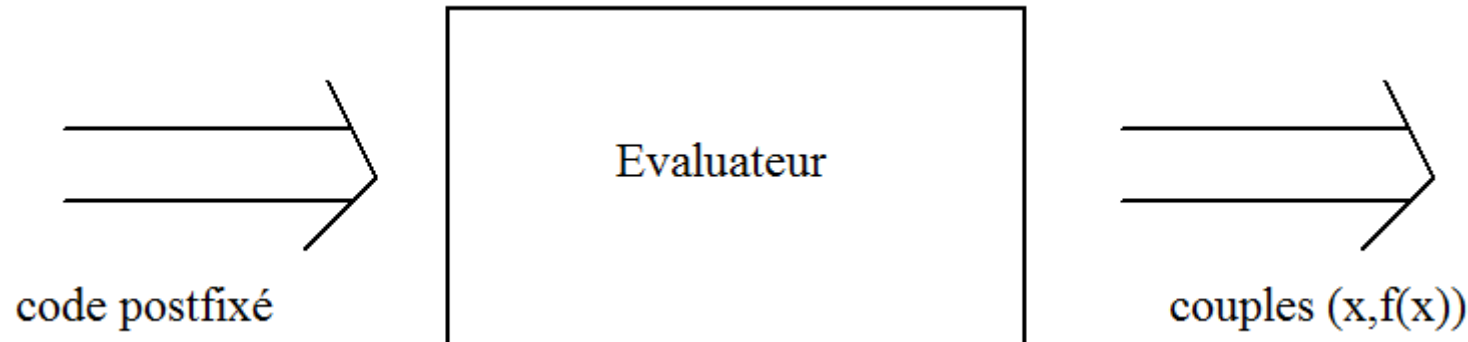
- Structure de l'arbre syntaxique:
  - Arbre binaire avec des jetons comme valeurs de nœud
  - Allocation de l'arbre:

```
Arbre noeud_arbre=(Arbre)malloc(sizeof(Node));  
noeud_arbre->pjeton_preced=NULL;  
noeud_arbre->pjeton_suiv=NULL;
```
  - En fonction de règle:
    - variable ou réel:
    - fonction
    - opérateur

# Réalisation d'un *grapheur d'expressions fonctionnelles*

## ❑ *Evaluateur*

- **Entrée** : code postfixé (ou arbre)
- **Sortie** : couples  $(x, f(x))$

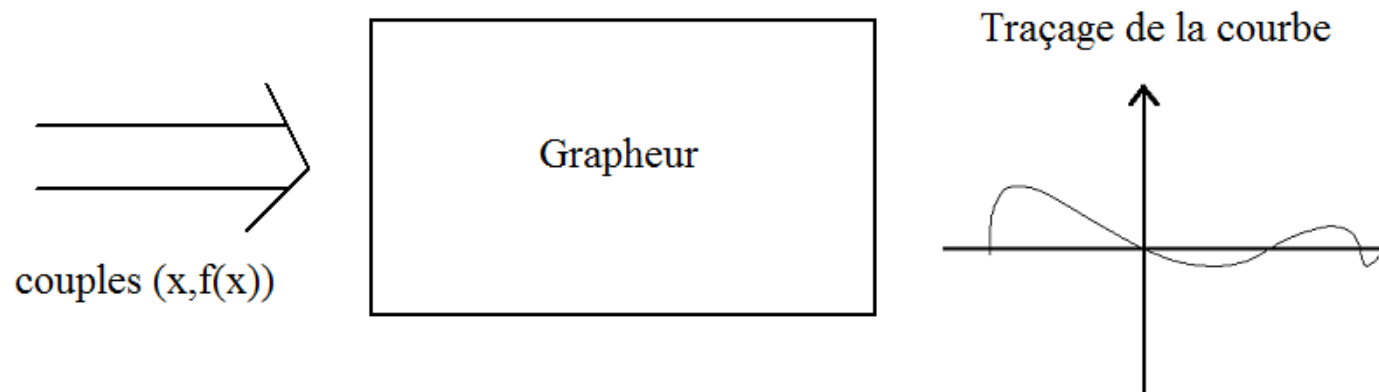


- Conception de la pile pour stocker le code postfixé
- Son rôle est de traiter le code postfixé produit en sortie par l'analyseur syntaxique (ou parcourir l'arbre) et de permettre la production des couples  $(x, f(x))$  à destination du grapheur

# Réalisation d'un *grapheur d'expressions fonctionnelles*

## □ Grapheur

- **Entrée** : couples  $(x, f(x))$
- **Sortie** : traçage de la courbe



- Chercher une liste de couples  $(x, f(x))$  et prendre les décisions nécessaires à son affichage (choix d'échelle, interface utilisateur, graduations, axes...)
- Gérer toutes les formes d'interaction entre l'utilisateur et l'ordinateur

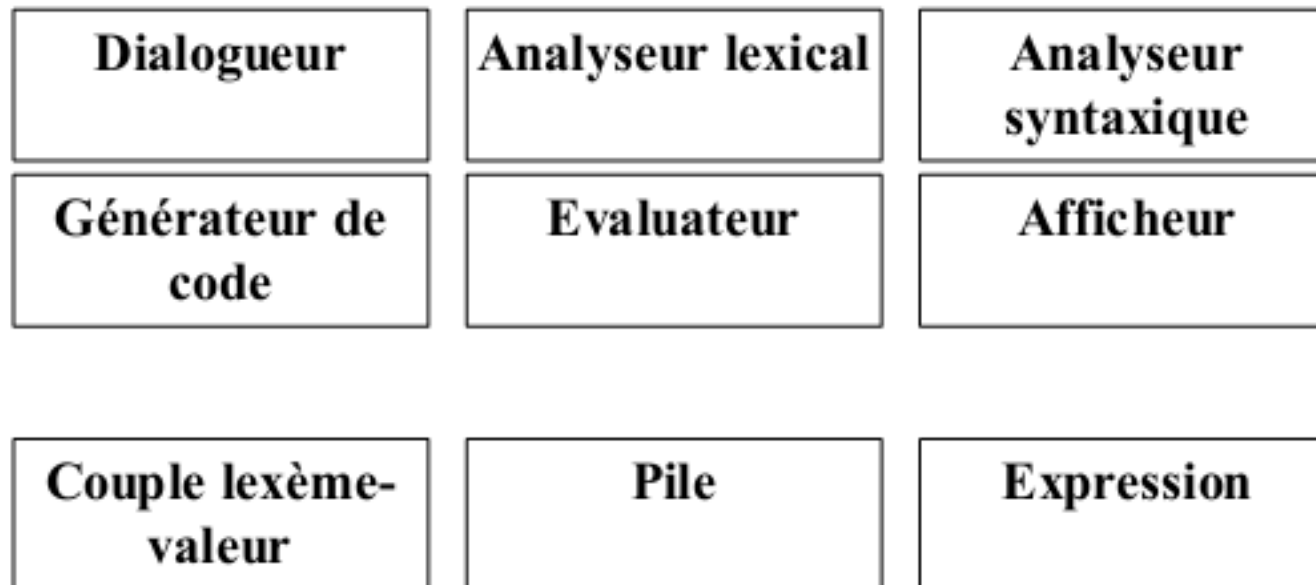
# Unité de compilation séparée

- Chaque module : couple fichier d'extension c et h
- Certains modules communiquent entre eux par appels de procédures,
- D'autres en partageant des structures de données communes
- Attention ! à la définition de l'interface (le fichier header) et de la visibilité réciproque de chacune de ces unités (chef de projet)
- Tester chaque module séparément : programme principal de test + minimum de structures de données

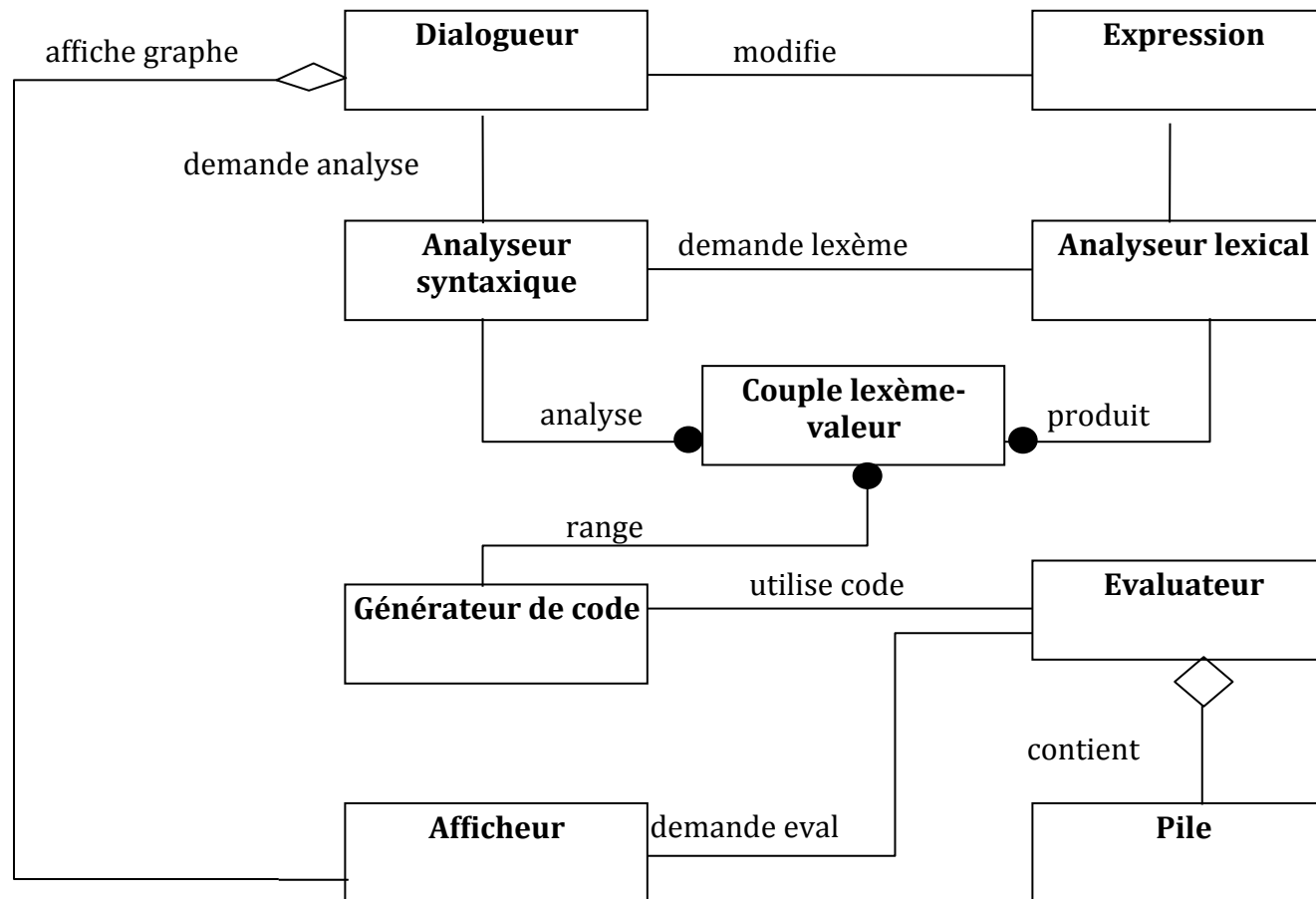
# Cahier de charges de l'application

- Développer une application qui permette à un utilisateur d'afficher la représentation graphique d'une fonction quelconque :
- Exprimer en utilisant :
  - les opérateurs arithmétiques courants  $+$ ,  $-$ ,  $*$ ,  $/$
  - les fonctions de base usuelles (abs, sin, sqrt, log etc..)
- Le parenthésage est possible sans limitation
- Les expressions comporteront une variable (notée  $x$  ou  $X$  )
- Sans oublier les bornes d'études de la fonction!
- La gestion des erreurs (division par zéro, racine de réel négatif etc...)
- Possibilité de modifier commodément l'expression / l'intervalle d'étude en cas d'erreur
- D'autres fonctionnalités facultative peuvent être fournies : zoom etc...

# Modélisation par objets



# Identifications des associations



# Modélisation fonctionnelle

