

*Programmation*  
*L1/ESI*  
*MALO Sadouanouan*

# Objectif

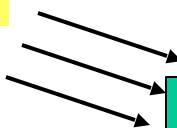
- Familiariser les étudiants avec les techniques et outils permettant :
  - de concevoir et de comprendre,
  - de réaliser puis traduire
  - d'implémenter et enfin d'obtenir un programme produisant par son exécution sur un ordinateur, le résultat attendu.

# 1. Introduction

- Algorithme :
  - suite des actions à effectuer pour
    - réaliser un traitement donné
    - résoudre un problème donné
- Exemples d'algorithme dans la vie courante
  - pour tricoter un pull : (maille à l'endroit, ...)
  - pour faire la cuisine : recette
  - pour jouer une sonate : partition

# 1.1 Algorithme

Informations  
en entrée



Algorithme informatique  
=  
procédure de calcul

Rigueur scientifique IMPORTANT !  
Sinon, information de sortie erronée



Informations  
en sortie

## 1.2 - Programmes

- Programme :
  - codage d'un algorithme afin que l'ordinateur puisse exécuter les actions décrites
  - doit être écrit dans un langage **compréhensible** par l'ordinateur
    - → langage de programmation (Assembleur (micropro), C, Fortran, Pascal, Python...)
- Un programme est donc une suite ordonnée d'instructions élémentaires codifiées dans un **langage** de programmation

# 1.3 - Langages de programmation

- L'ordinateur
  - construit autour d'un ensemble de circuits électroniques (le courant passe, le courant ne passe pas)
  - traite donc que des signaux assimilables à 0 ou 1
  - une opération élémentaire → suite de 0 et de 1 = suite de bits (BInary digiT) ! Un champ de 8 bits constituant ce qu'on appelle 1 byte ou 1 octet. Importance des unités en science. Rappel:  $k(2^{10})$  M et G .
- Pour que les programmes et les données soient compréhensibles par l'ordinateur il faut effectuer un codage binaire

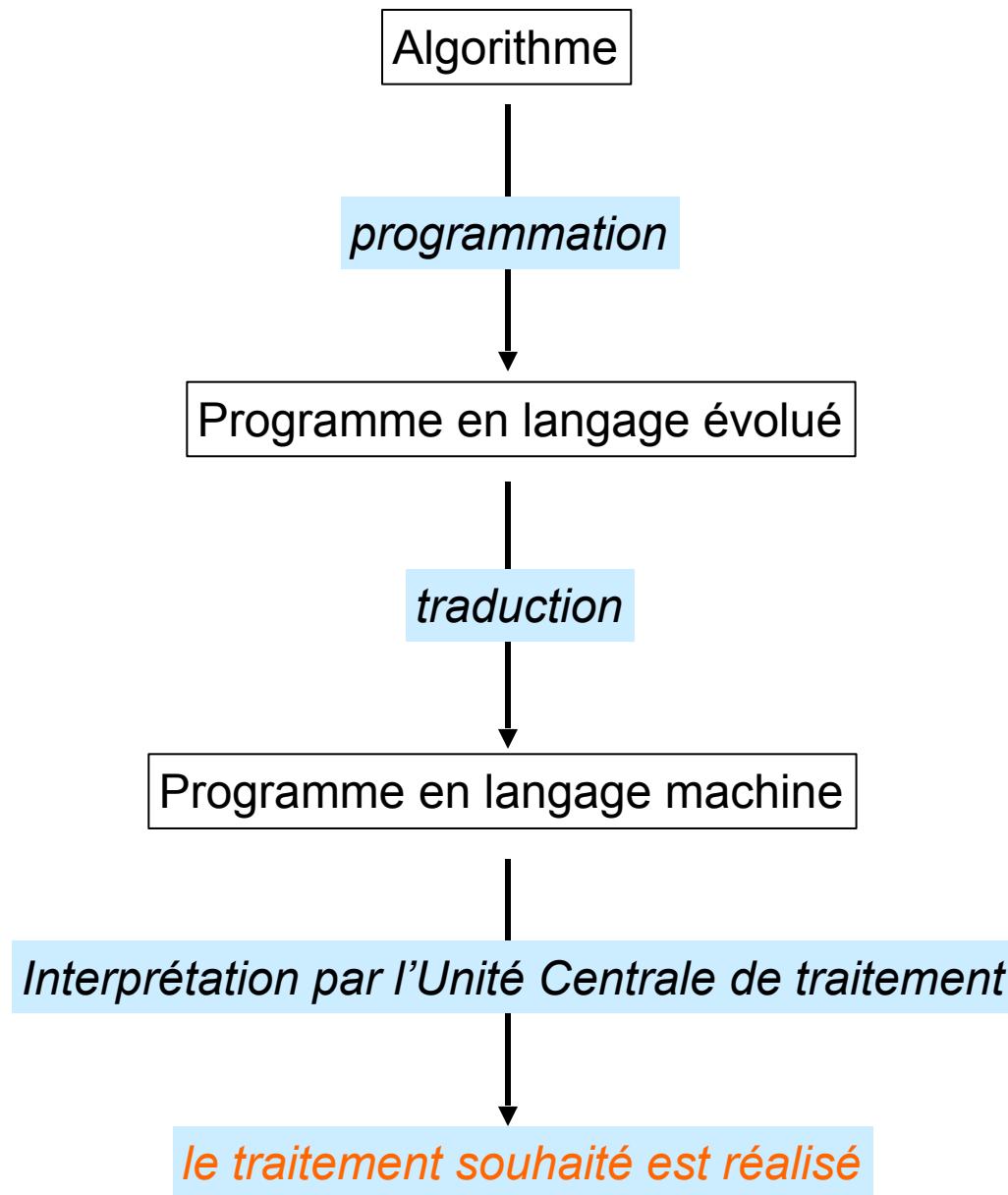
# 1.3 - Langages de programmation

- Langage machine
  - langage binaire
  - ses opérations sont directement compréhensibles par l'ordinateur
  - **propre à chaque famille d'ordinateur**
  - *Pour pouvoir manipuler du langage machine, on est obligé de passer par de l'Assembleur.*
- Ecriture des premiers programme en langage machine

# 1.3 - Langages de programmation

**TYPES DE LANGAGES → STYLE DE PROGRAMMATION**

- langages impératifs (Fortran, Pascal, C ...) : Il s'agit de faire exécuter une suite d'ordres par une machine bête mais disciplinée.
- langages Déclaratifs: l'activité de programmation consiste essentiellement à décrire le *rappor*t qui existe entre les données et les résultats que l'on veut obtenir, plutôt que la séquence de traitements qui mène des unes aux autres
  - fonctionnels (Lisp, Scheme ...)
  - logiques (Prolog ... )
- langages objets (C++, VisualBasic, Delphi, Java ...)



## 1.4-Exemple d'algorithme

- Calcul de l'intérêt et de la valeur acquise par une somme placée pendant un an
- L'énoncé du problème indique
  - Les données fournies: deux nombres représentant les valeurs de la somme placée et du taux d'intérêt
  - les résultats désirés: deux nombres représentant l'intérêt fourni par la somme placée ainsi que la valeur obtenue après placement d'un an.

## 1.4-Exemple d'algorithme

- Formalisation de l'algorithme:En français
  1. prendre connaissance de la somme initiale et du taux d'intérêt
  2. multiplier la somme par le taux; diviser ce produit par 100; le quotient obtenu est l'intérêt de la somme
  3. additionner ce montant et la somme initiale; cette somme est la valeur acquise
  4. afficher les valeurs de l'intérêt et de la valeur acquise.
  - SI=somme initiale
  - T=taux d'intérêt (ex: 3 pour 3%)
  - I=intérets=S\*T/100
  - SF=somme finale=S+I

## 1.4-Exemple d'algorithme

- Formalisation de l'algorithme
  - En langage de description : pseudo code, LDA (Langage de Description Algorithmique)

écrire " Introduisez la somme initiale (en francs): "

Lire SI

écrire " Introduisez le taux d'intérêt (ex: 3 pour 3%): "

lire T

$I \leftarrow SI * T / 100$

$SF \leftarrow SI + I$

écrire " L'intérêt fourni est de " , I , "francs "

écrire " La somme après un an sera de " , SF , "francs "

## 2-Programmation en langage C

## 2.1 - Instructions d'entrée/sortie et déclaration de variables

- Exemple :
  - Exprimer un nombre exprimé sous forme de secondes en « heures », « minutes » et « secondes ».
  - La seule donnée est le nombre total de secondes que nous appellerons **nsec** ; les résultats consistent en 3 nombres : **h, m, s**.

## 2.1 - Instructions d'entrée/sortie et déclaration de variables

**écrire** " Introduisez le nombre de secondes"

**lire** nsec

s  $\leftarrow$  nsec mod 60

m  $\leftarrow$  (nsec div 60) mod 60

h  $\leftarrow$  nsec div 3600

**écrire** nsec, "valent: ", h, "heure(s) ", m, "minute(s) et", s,  
« seconde(s)"

## 2.1 - Instructions d'entrée/sortie et déclaration de variables

- Il est aussi nécessaire de préciser ce que les variables utilisées contiendront comme type de données.
- Il peut s'agir de nombres entiers, de nombres réels, de chaînes de caractères, ...
- Il faut faire précéder la description de l'algorithme par une partie dite **déclarative** où l'on regroupe les **caractéristiques** des variables manipulées.

## 2.1 - Instructions d'entrée/sortie et déclaration de variables

- La partie déclarative est placée (généralement) en tête de l'algorithme et regroupe une ou plusieurs indications de la forme:

**entier variables**

ou

**décimaux (réel) variables**

## 2.1 - Instructions d'entrée/sortie et déclaration de variables

entier nsec, h, m, s

écrire " Introduisez le nombre de secondes"

lire nsec

s  $\leftarrow$  nsec mod 60

m  $\leftarrow$  (nsec div 60) mod 60

h  $\leftarrow$  nsec div 3600

écrire nsec, "valent: ", h, "heure(s) ", m, "minute(s) et", s,  
"seconde(s)"

## 2.1 - Instructions d'entrée/sortie et déclaration de variables

- Un identificateur (nom de variable, fonction, ...) en C doit débuter par une lettre suivie par un nombre quelconque de lettres, chiffres ou de "\_" (caractère souligné).
- Les identificateurs ne peuvent contenir d'espacement (caractère "blanc") ou de caractères tels que %, ?, \*, ., - ,... mais peuvent être aussi longs que l'on veut.

## 2.1 - Instructions d'entrée/sortie et déclaration de variables

- Les variables doivent faire l'objet d'une déclaration de type de la forme:  
`type liste_des_variables ;`
- Des points-virgules sont obligatoires pour séparer les instructions
- Les instructions de lecture et d'écriture se traduisent respectivement par **SCANF** et **PRINTF** suivis d'une liste de variables ou d'expressions placées entre parenthèses et séparées par des virgules.

## 2.1 - Instructions d'entrée/sortie et déclaration de variables

```
#include <stdio.h>                      décrit les fonctions de lecture et affichage sur l'écran
int nsec, h, m, s;                        déclarations
main(){                                      marque le début du programme principal
    printf("Introduisez le nombre en secondes : ");
    scanf("%d", &nsec);
    s=nsec % 60;
    m=nsec / 60 % 60;
    h=nsec / 3600;
    printf("%d secondes valent %d heures %d minutes et %d
           secondes\n",nsec,h,m,s);
}
```

*marque la fin du programme principal*

## 2.2 - Structures de contrôle

- Il a été démontré que pour représenter n'importe quel algorithme, il faut disposer des trois possibilités suivantes:
  - La structure de **séquence** qui indique que les opérations doivent être exécutées les unes après les autres
  - la structure de **répétition** qui indique qu'un ensemble d'instructions doit être exécuté plusieurs fois.
  - la structure de **choix (alternative)** qui indique quel ensemble d'instructions doit être exécuté suivant les circonstances

## 2.2 - Structures de contrôle: la structure alternative

- Exemple :
  - 2 joueurs A et B
  - Chacun montre un certain nombre de doigts (de 0 à 5)
  - Si la somme des nombres de doigts montrés est paire, le premier joueur a gagné
  - Sinon c'est le second.
  - Le problème est de faire prendre la décision par l'ordinateur.

## 2.2 - Structures de contrôle: la structure alternative

- En Français :
  - prendre connaissance du nombre de doigts de A
  - prendre connaissance du nombre de doigts de B
  - calculer la somme de ces deux nombres
  - si la somme est paire, A est le gagnant
  - si la somme est impaire, B est le gagnant.
- Remarque: Pour déterminer si un nombre est pair ou impair, il suffit de calculer le reste de la division par 2 (.. modulo 2): il vaut 0 dans le premier cas et 1 dans le second.

## 2.2 - Structures de contrôle: la structure alternative

- En LDA :

entier na,nb,reste

lire na,nb

reste  $\leftarrow$ (na + nb) mod 2

si reste = 0 alors écrire "Le joueur A a gagné."

sinon écrire "Le joueur B a gagné."

fsi

écrire « Fin du jeu »

## 2.2 - Structures de contrôle: la structure alternative

- La structure alternative se présente en général sous la forme :

**si** expression **alors**

première séquence d'instructions

**sinon**

deuxième séquence d'instructions

**fsi**

## 2.2 - Structures de contrôle: la structure alternative

- où **expression** conditionne le choix d'un des deux ensembles d'instructions. Cette **expression** peut être soit vraie soit fausse
- Si l'**expression** est vraie, la **première séquence d'instruction** sera exécutée et la seconde sera ignorée;
- Si l'**expression** est fausse, seule la **seconde séquence d'instructions** sera effectuée.

## 2.2 - Structures de contrôle: la structure alternative

- Le mot **sinon** indique où se termine la première séquence d'instructions et où commence la seconde.
- Le mot **fsi** (abrégé de "fin de si") indique où se termine la seconde séquence d'instructions.

## 2.2 - Structures de contrôle: la structure alternative

- Dans certains cas, lorsque l'expression est fausse, aucune instruction ne doit être exécutée. La condition s'exprime alors plus simplement sous la forme:

**si expression alors**

séquence d'instructions

**fsi**

- Quoi qu'il arrive, les instructions qui suivent **fsi** seront exécutées.
- Chacune des séquences d'instructions d'un **si ... fsi** peut contenir des **si...fsi**. On dit alors que les structures sont **imbriquées**.

## 2.2 - Structures de contrôle: la structure alternative

- Prendre l'habitude de décaler et d'utiliser les **fsi**  
**si** expression1 **alors**  
    **si** expression2 **alors**  
        instruction1  
    **sinon**  
        instruction2
- Différent de ...

## 2.2 - Structures de contrôle: la structure alternative

<u>si</u> expression <u>alors</u> séquence d'instructions <u>fsi</u>	<b>if</b> (expression) { séquence_d_instructions; }
<u>si</u> expression <u>alors</u> une instruction <u>sinon</u> séquence d'instructions <u>fsi</u>	<b>if</b> (expression) une_instruction; <b>else</b> { séquence_d_instructions; }

### Remarque

- En C, l'expression est un nombre 0 ou 1

## 2.2 - Structures de contrôle: la structure alternative

```
#include <stdio.h>

int NA, NB, reste;
main () {
    printf("Introduisez le nombre de doigts montrés par le joueur A : ");
    scanf("%d",&NA);
    printf("Introduisez le nombre de doigts montrés par le joueur B : ");
    scanf("%d",&NB);
    reste = (NA+NB) % 2;
    if (reste==0)
        printf("Le joueur A a gagné\n");
    else
        printf("Le joueur B a gagné\n");
    printf("Bravo pour le gangnant\n");
}
```

## 2.2 - Structures de contrôle: la structure alternative

### Expression logique en C

- <    >
- ==
- !=
- >=   <=
- &&
- ||

## 2.3 - Structures de contrôle: la structure répétitive

- L'utilisation d'un ordinateur s'impose lorsque des volumes importants de données sont manipulées
- **Exemple 1:** Chercher dans une liste de noms et d'adresses, l'adresse d'une personne à partir de son nom. Le nombre de fois qu'il faudra comparer le nom donné aux noms de la liste est dans ce cas inconnu
- **Exemple 2 :** Calculer la N<sup>ème</sup> puissance entière d'un nombre x par multiplications successives du nombre par lui-même. Ici, le nombre de répétition (N) de l'instruction de multiplication est connu.

## 2.3.1 - Structures de contrôle: la structure répétitive " pour faire"

- Lorsque le nombre d'itération est connu
- Exemple de la table de multiplication
- En LDA :

**pour var\_de\_crt  $\leftarrow$  prem\_val à dern\_val faire**  
**séquence d'instructions**  
**fpour**

### 2.3.1 - Structures de contrôle: la structure répétitive " pour faire"

- **var\_de\_crt** : est la variable de contrôle, qui est initialisée à **prem\_val**
- La variable de contrôle est incrémenté automatiquement à chaque itération
- L’itération s’arrête lorsque :  
**var\_de\_crt > dern\_val**
- Si **prem\_val > dern\_val** la **séquence d’instructions** n’est jamais exécutée
- La variable de contrôle doit être de type énuméré
- La valeur de la variable de contrôle ne doit pas être directement modifiée dans la **séquence d’instructions**

## 2.3.1 - Structures de contrôle: la structure répétitive " pour faire"

- Écrire l'algorithme qui affiche la table de multiplication de 6.

**entier** n

**lire** n

**Pour** i  $\leftarrow$  1 **à** 10 **faire**

**écrire** n, « fois », i, « font », n\*i

**fpour**

### 2.3.1 - Structures de contrôle: la structure répétitive " pour faire"

- En C, la boucle pour se traduit par

```
for (exp_init ; exp_cond ; exp_evol)  
    <instruction;>
```

- Ou

```
for (exp_init ; exp_cond ; exp_evol) {  
    <séquence d'instructions;>  
}
```

## 2.3.1 - Structures de contrôle: la structure répétitive " pour faire"

- **exp\_init** :
  - est une instruction d'initialisation ; elle est exécutée avant l'entrée dans la boucle
- **exp\_cond** :
  - est la condition de continuation ; elle est testée à chaque passage, y compris lors du premier ; l'instruction ou les instructions composant le corps du for sont répétées tant que le résultat de l'expression **exp\_cond** est VRAI
- **exp\_evol** :
  - Est une instruction de rebouclage ; elle fait avancer la boucle ; elle est exécutée en fin de boucle avant le nouveau test de passage.

## 2.3.1 - Structures de contrôle: la structure répétitive "pour faire"

- **Exemple 3:** afficher la table de multiplication à l'ancienne :
  - 6 fois 1 font 6
  - 6 fois 2 font 12
  - ...
  - 6 fois 10 font 60
- Il serait plus judicieux de faire répéter la ligne d'affichage en faisant varier le multiplicande
  - Pour chaque valeur de  $n$  variant de 1 à 10 exécuter :  
Ecrire « 6 fois »,  $n$ , « font »,  $6*n$

## 2.3.1 - Structures de contrôle: la structure répétitive " pour faire"

```
#include <stdio.h>
main(){
    int i,n;
    printf("Quelle table souhaitez-vous afficher ? : ");
    scanf("%d",&n);
    for (i=1;i<=10;i++)
        printf("%d fois %d font %d\n",n,i,n*i);
}
```

++ : opérateur d'incrémentation

$i++ \Leftrightarrow i=i+1$

-- : opérateur de décrémentation

$i-- \Leftrightarrow i=i-1$

## 2.3.1 - Structures de contrôle: la structure répétitive " pour faire"

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

```
main() {  
    int i;  
    for (i=1;i<=5;i++)  
        printf("%d : On incrémente\n",i);  
    printf("-----\n");  
    for (i=5;i>=1;i--)  
        printf("%d : On décrémente\n",i);  
}
```

## 2.3.1 - Structures de contrôle: la structure répétitive " pour faire"

1 : On incrémente  
2 : On incrémente  
3 : On incrémente  
4 : On incrémente  
5 : On incrémente

---

5 : On décrémente  
4 : On décrémente  
3 : On décrémente  
2 : On décrémente  
1 : On décrémente

## 2.3.2 - Structures de contrôle: la structure répétitive " tant que "

lire nom\_donné

lire nom1

si nom1 = nom\_donné alors

écrire adresse1

sinon

lire nom2

si nom2 = nom\_donné alors

écrire adresse2

sinon

lire nom3

si nom3 = nom\_donné alors ...

## 2.3.2 - Structures de contrôle: la structure répétitive "tant que"

- Fait répéter une séquence d'instructions aussi longtemps qu'une condition est VRAI
- En LDA :

**Tant que **condition faire**  
**séquence d'instructions****

**Ftq**

## 2.3.2 - Structures de contrôle: la structure répétitive "tant que"

- Au moment du premier passage dans la boucle la **condition** est évaluée; si elle est vérifiée, la **séquence d'instructions** est exécutée
- A la fin de l'exécution de cette **séquence d'instructions**, la **condition** est de nouveau évaluée et on répète l'exécution de la **séquence d'instructions** tant que la **condition** est vérifiée.
- Dès que la **condition** devient fausse, l'exécution du programme se poursuit à partir de la 1<sup>ère</sup> instruction qui suit immédiatement le mot-clé **ftq**.

## 2.3.2 - Structures de contrôle: la structure répétitive "tant que"

- Remarques :
  - Si au premier passage la **condition** est évaluée à FAUX, le corps de la boucle (la **séquence d'instructions**) n'est jamais exécuté
  - Si la **séquence d'instructions** ne change pas la valeur de la **condition**, la **séquence d'instructions** sera exécutée sans que l'on passe jamais à la suite : on exécute une **boucle infinie**
  - Les variables qui interviennent dans la **condition** doivent être initialiser avant d'aborder la boucle

## 2.3.2 - Structures de contrôle: la structure répétitive " tant que "

- Remarques :
  - Il est préférable d'exprimer l'expression logique sous la forme NON(**condition(s) d'arrêt**).
  - Il est en effet plus simple de déterminer les raisons d'arrêter le processus répétitif que celles de continuer.
  - La forme de ce type de boucle devient donc :

**tant que NON **condition(s) d'arrêt faire**  
**séquence d'instructions****  
**ftq**

## 2.3.2 - Structures de contrôle: la structure répétitive "tant que"

lire nom\_donné

i  $\leftarrow$  1

lire nomi

tant que NON ((nomi = nom\_donné) ou (*fin de liste*)) faire

i  $\leftarrow$  i+1

lire nomi

ftq

si nomi = nom\_donné alors

écrire adressei

sinon

écrire "Le nom demandé ne se trouve pas dans la liste."

fsi

## 2.3.2 - Structures de contrôle: la structure répétitive "tant que"

- Écrire l'algorithme qui affiche la table de multiplication de 6.

entier n

lire n

i  $\leftarrow$  1

tant que NON (i > 10) faire e

écrire n, « fois », i, « font », n\*i

    i  $\leftarrow$  i+1

ftq

## 2.3.2 - Structures de contrôle: la structure répétitive "tant que"

- Considérons aussi l'exemple suivant :
  - étant donnés deux nombres entiers **m** et **n** positifs ou nuls, calculer le PGCD.
  - L'algorithme d'Euclide permet de résoudre ce problème :
    - en prenant d'abord le reste de la division de **m** par **n**,
    - puis le reste de la division de **n** par ce premier reste,
    - etc.
    - jusqu'à ce qu'on trouve un reste nul.
    - Le dernier diviseur utilisé est le PGCD de **m** et **n**.

## 2.3.2 - Structures de contrôle: la structure répétitive "tant que"

- Pour **m**=1386 et **n**=140, on a successivement :
  - $1386 = 140 * 9 + 126$
  - $140 = 126 * 1 + 14$
  - $126 = 14 * 9 + 0$
- et le PGCD de 1386 et 140 est bien 14.
- Dans cet exemple, il faut répéter le calcul du reste de la division d'un nombre par un autre.
- Appelons **a** le dividende, **b** le diviseur et **r** le reste

## 2.3.2 - Structures de contrôle: la structure répétitive "tant que"

**entier** m,n,a,b,r,PGCD

**Début**

**lire** m, n

a  $\leftarrow$  m

b  $\leftarrow$  n

**tant que** NON(b = 0) **faire**

r  $\leftarrow$  a mod b

a  $\leftarrow$  b

b  $\leftarrow$  r

**ftq**

PGCD  $\leftarrow$  a

**écrire** "Le PGCD de",m,"et",n,"est",PGCD

**Fin**

## 2.3.2 - Structures de contrôle: la structure répétitive "tant que"

- En C, la boucle tant que se traduit par **WHILE** ( <expression logique>) **instruction**;

- ou

```
WHILE ( <expression logique> ) {  
    séquence d'instructions;  
}
```

## 2.3.2 - Structures de contrôle: la structure répétitive "tant que"

Exemple : Algorithme Euclide (PGCD)

- Soit  $r$  le reste de la division euclidienne de  $a$  par  $b$  :  
$$a = bq + r, \quad r < b.$$
- Tout diviseur commun de  $a$  et  $b$  divise aussi  $r = a - bq$ , et réciproquement tout diviseur commun de  $b$  et  $r$  divise aussi  $a = bq + r$ . Donc le calcul du PGCD de  $a$  et  $b$  se ramène à celui du PGCD de  $b$  et  $r$  ; et on peut recommencer sans craindre une boucle sans fin, car les restes successifs sont strictement décroissants. Le dernier reste non nul obtenu est le PGCD cherché.

## 2.3.2 - Structures de contrôle: la structure répétitive " tant que "

```
#include <stdio.h>
int m,n,a,b,r,PGCD;
main(){
    printf("Nous allons calculer le PGCD de 2 nombres\n");
    printf("Introduisez le premier nombre : ");
    scanf("%d",&m);
    printf("Introduisez le second nombre : ");
    scanf("%d",&n);
    a=m; b=n;
    while (b!=0) {
        r=a % b;
        a=b; b=r;
    }
    PGCD=a;
    printf("Le PGCD de %d et %d est %d\n",m,n,PGCD);
}
```

Exemple : Algorithme Euclide (PGCD)

a=60	b=42
r=18	
42+18	18
42	18
2x18+6	18
6	18
6	6x3
a=6	b=0

## 2.3.2 - Structures de contrôle: la structure répétitive "tant que"

- Somme des 10 premiers entiers : comparaison entre l'utilisation et de la boucle « **tant que** » de la boucle « **pour** »

```
somme = 0;  
i=0;  
while (i<10) {  
    somme = somme + i;  
    i = i + 1;  
}
```

```
somme = 0;  
for (i=0;i<10;i++)  
    somme = somme + i;
```

## 2.3.2 - Structures de contrôle: la structure répétitive " tant que "

- Le langage C propose également une autre forme de la boucle tant que qui permet d'exécuter au moins une fois le corps de la boucle :

DO

<instruction;>

WHILE ( <expression logique>);

- ou

DO {

<séquence d'instructions;>

} WHILE ( <expression logique>);

## 2.3.2 - Structures de contrôle: la structure répétitive "tant que"

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

```
main() {
    int i, somme, N;
    somme=0;
    printf("Entrez le nombre d'élément que vous voulez sommer : ");
    scanf("%d",&N);
    i=1;
    while (i<N) {
        somme = somme +i;
        i=i+1;
    }
    printf("Somme des %d premiers entiers est :
    %d\n",N,somme);
}
```

## 2.3.2 - Structures de contrôle: la structure répétitive "tant que"

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

```
main() {
    int i, somme, N;
    somme=0;
    printf("Entrez le nombre d'élément que vous voulez sommer : ");
    scanf("%d",&N);
    i=1;
    do {
        somme = somme +i;
        i=i+1;
    } while (i<N);
    printf("Somme des %d premiers entiers est : %d\n",N,somme);
}
```

## 2.4 - Structures de contrôle: Le choix multiple

- Supposons que l'on veuille demander à l'utilisateur de choisir dans un menu une des 3 possibilités offertes.
- Le choix présenté ne se limite pas à une alternative (soit - soit).
- Mais plutôt à une expression du type « selon que... »

## 2.4 - Structures de contrôle: Le choix multiple

- En LDA :

entier i

lire i

selon que

i=1 faire bloc1

ou que i=2 faire bloc2

ou que i=3 faire bloc3

autrement écrire "Mauvais choix"

Fselon

- autrement est comme dans l'alternative facultative

## 2.4 - Structures de contrôle: Le choix multiple

- Peut toujours s'écrire avec des alternatives :

entier i

lire i

Si i=1 alors

bloc1

sinon

si i=2 alors

bloc2

sinon

si i=3 alors

bloc3

sinon

écrire "Mauvais choix"

Fsi

Fsi

Fsi

## 2.4 - Structures de contrôle: Le choix multiple

- La traduction du choix multiple en C est assez restrictive, puisque la valeur de l'expression conditionnant le choix doit être entière (**char**, **short**, **int**).
- L'instruction **switch** permet de mettre en place une structure d'exécution qui permet des choix multiples parmi des cas de même type et faisant intervenir uniquement des **valeurs constantes entières**.

## 2.4 - Structures de contrôle: Le choix multiple

Switch ( <expression entière> ) {

    case <constante entière> :

        <instruction 1>

        ...

        <instruction N>

        ...

    default :

        <instruction 1>

        ...

        <instruction N>

}

ATTENTION : si la dernière instruction n'est pas l'instruction **break** les autres « **case** » ainsi que le « **default** » seront exécutés

## 2.4 - Structures de contrôle: Le choix multiple

```
#include <stdio.h>
int i;
main() {
    printf("Entrez votre choix : ");
    scanf("%d",&i);
    switch(i) {
        case 1:printf("premier choix\n");
                  break;
        case 2:printf("deuxième choix\n");
        case 3:printf("troisième choix\n");
        default:printf("Autre choix que choix 1, 2 ou 3\n");
    }
}
```

Entrez votre choix : 1  
premier choix

Entrez votre choix : 2  
deuxième choix  
troisième choix  
Autre choix que choix 1, 2 ou 3

## 2.4 - Structures de contrôle: Le choix multiple

```
#include <stdio.h>
int i;
main() {
    printf("Entrez votre choix : ");
    scanf("%d",&i);
    switch(i) {
        case 1:printf("premier choix\n");
                  break;
        case 2:printf("deuxième choix\n");
                  break;
        case 3:printf("troisième choix\n");
                  break;
        default:printf("Autre choix que choix 1, 2 ou 3\n");
    }
}
```

# Transition

- Les variables utilisées jusqu'à présent sont parfois inadaptées au traitement à réaliser
  - Difficulté d'en utiliser un nombre important
  - Particulièrement lorsqu'il s'agit de valeurs de même type
  - Exemples : relevés mensuels, statistiques journalières

## 2.5 Les tableaux-Introduction

- Imaginons que nous ayons plusieurs traitements à effectuer sur des consommations mensuelles d'accès à Internet (en nombre de Mo)
  - Pour conserver les valeurs en mémoire, il est possible de prendre 12 variables numériques, `cons_Internet1`, `cons_Internet2`, ..., `cons_Internet12`.
  - Pour calculer la consommation mensuelle moyenne sur l'année, la formule suivante peut être écrite :

Moyenne  $\leftarrow$  (`cons_Internet1` + `cons_Internet2` + `cons_Internet3` +  
`cons_Internet4` + `cons_Internet5` + `cons_Internet6` + `cons_Internet7` +  
`cons_Internet8` + `cons_Internet9` + `cons_Internet10` + `cons_Internet11` +  
`cons_Internet12`) / 12

- Difficile à faire avec 365 consommations journalières...

## 2.5 Les tableaux-Introduction

- Imaginons que nous ayons à compter le nombre de fois, que chacune des notes, de 0 à 20, a été attribuée à une population d'un grand nombre d'étudiants.
  - L'algorithme pourrait être le suivant :

Initialiser les 21 variables à 0

Lire n

Tant que n ≠ 99 faire

si n=0 alors note0 ← note0 + 1

si n=1 alors note1 ← note1 + 1

...

si n=20 alors note20 ← note20 + 1

Ecrire « donnez la note suivante ou 99 pour terminer »

lire n

Ftq

## 2.5 Les tableaux-Introduction

- Pour simplifier cet algorithme, il suffirait de pouvoir désigner directement la note qui correspond à la valeur de la variable n
- La structure de données tableau regroupe une famille de variables et permet de résoudre ce genre de problème

## 2.5 Les tableaux-Tableaux à un indice

- Un tableau (encore appelé table ou variable indicée) est un ensemble de données, qui sont toutes de même type, désigné par un identificateur unique (le nom du tableau), et qui se distinguent les une des autres par leur numéro d'indice
- Exemple : les températures sous abri à 15h00 des jours d'une semaine seront les 7 valeurs de la variable température, qui est un tableau de 7 éléments (variables) de type réel désigné par :
  - Température[1], Température[2], ..., Température[7],

## 2.5 Les tableaux-Tableaux à un indice

- Représentation graphique :  
température

Nom du tableau

1	25
2	30
3	
4	36
5	33
6	22
7	27

Cette case du tableau  
représente la variable  
**Température[3]** dont la valeur est

## 2.5 Les tableaux-Tableaux à un indice

- L'utilisation d'un indice variable présente le principal intérêt des tableaux
- Si la variable  $k$  entière a pour valeur 3, alors
  - $\text{Température}[k] = 23$
  - $\text{Température}[k+1] = 36$
  - $\text{Température}[k-1] = 30$
- Mais **attention** l'écriture de  $\text{Température}[-2]$  ou  $\text{Température}[12]$  n'ont pas de sens car elles font référence à des éléments inexistantes

## 2.5 Les tableaux-Tableaux à un indice

- De même le calcul de la température moyenne de la semaine se fera de façon très simple :

**Entiers** somme, k

**Réel** moyenne

somme  $\leftarrow$  0

**Pour** k  $\leftarrow$  1 **à** 7 **faire**

somme  $\leftarrow$  somme + Température[k]

**fpour**

moyenne  $\leftarrow$  somme / 7

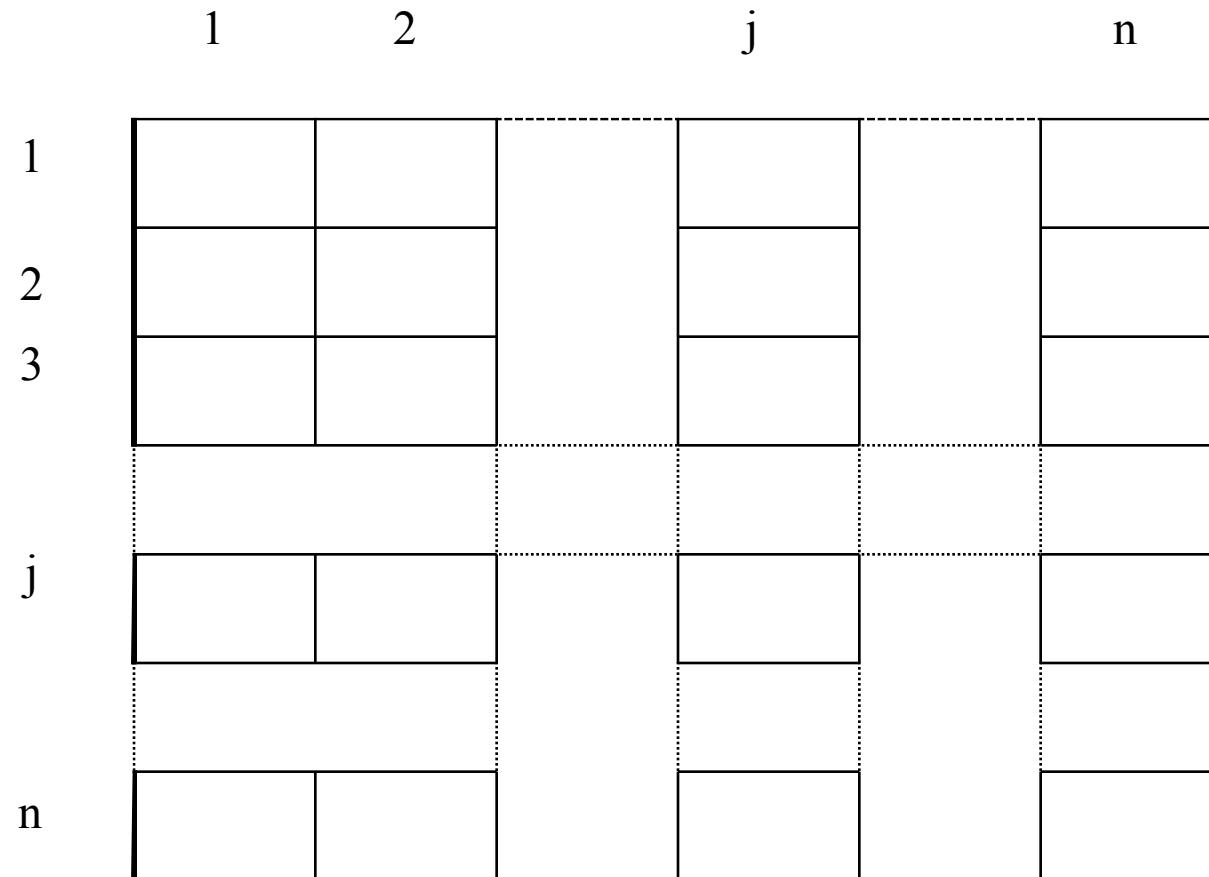
## 2.5 Les tableaux-Tableaux à un indice

- Comme tout objet un tableau doit être déclaré avant tout utilisation
- 3 éléments fondamentaux définissent un tableau à un indice :
  - Son nom : identificateur respectant les règles classiques des identificateurs d'un programme
  - Le nombre de ses éléments
  - Le type de données qu'il contient

Tableau de 7 entiers Température

Tableau de [1..7] d'entiers Température

## 2.5 Les tableaux-Tableaux à plusieurs indices



## 2.5 Les tableaux-Tableaux en C

- Exemple :

```
int a[13];
char b[8][5][10];
float d [6][15][9];
```

- Principe :

<type><identificateur>[taille<sub>1</sub>][taille<sub>2</sub>]...[taille<sub>k</sub>];

- N'importe quelle référence à une case peut être utilisé comme une simple variable :

```
int i,j,k;
a[i]
b[i][j][k]
```

## 2.5 Les tableaux-Tableaux en C

- La **taille** correspond au nombre de cases du tableau
- Attention : les indices, permettant de localiser le contenu d'une case d'un tableau, varient entre 0 et **taille-1**
- Il est possible d'affecter un tableau à un ensemble de valeurs dès sa déclaration par :  
`<type><identificateur>[taille1][taille2]...={val1, val2, ...};`
- Exemple :

`Int matrice[2][3]={1,2,3,4,5,6} ⇔`

1	2	3
4	5	6

## 2.5 Les tableaux-Tableaux en C

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

```
main(){
    int i, somme, temperature[7];
    float moyenne;
    for (i=0;i<7;i++){
        printf("Temperature[%d]=",i);
        scanf("%d",&temperature[i]);
    }
    somme=0;
    for (i=0;i<7;i++)
        somme=somme+temperature[i];
    moyenne=somme/7;
    printf("la température moyenne de la semaine est %f\n",moyenne);
}
```

## 2.6 Les sous-programmes-Introduction

- Jusqu'ici nous avons étudié les algorithmes selon une approche simple :
  - Un seul algorithme exprime une fonction
  - qui à partir de données initiales produit un résultat
- Cas plus général :
  - Le calcul d'une fonction est le résultat de plusieurs algorithmes disjoints
  - À chaque algorithme va donc correspondre un programme
  - L'exécution restera quand même séquentiel  $\Rightarrow$  un seul programme s'exécute et exécute l'une de ses instructions

## 2.6 Les sous-programmes-Introduction

### Exemple simple d'appels de sous-programmes

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

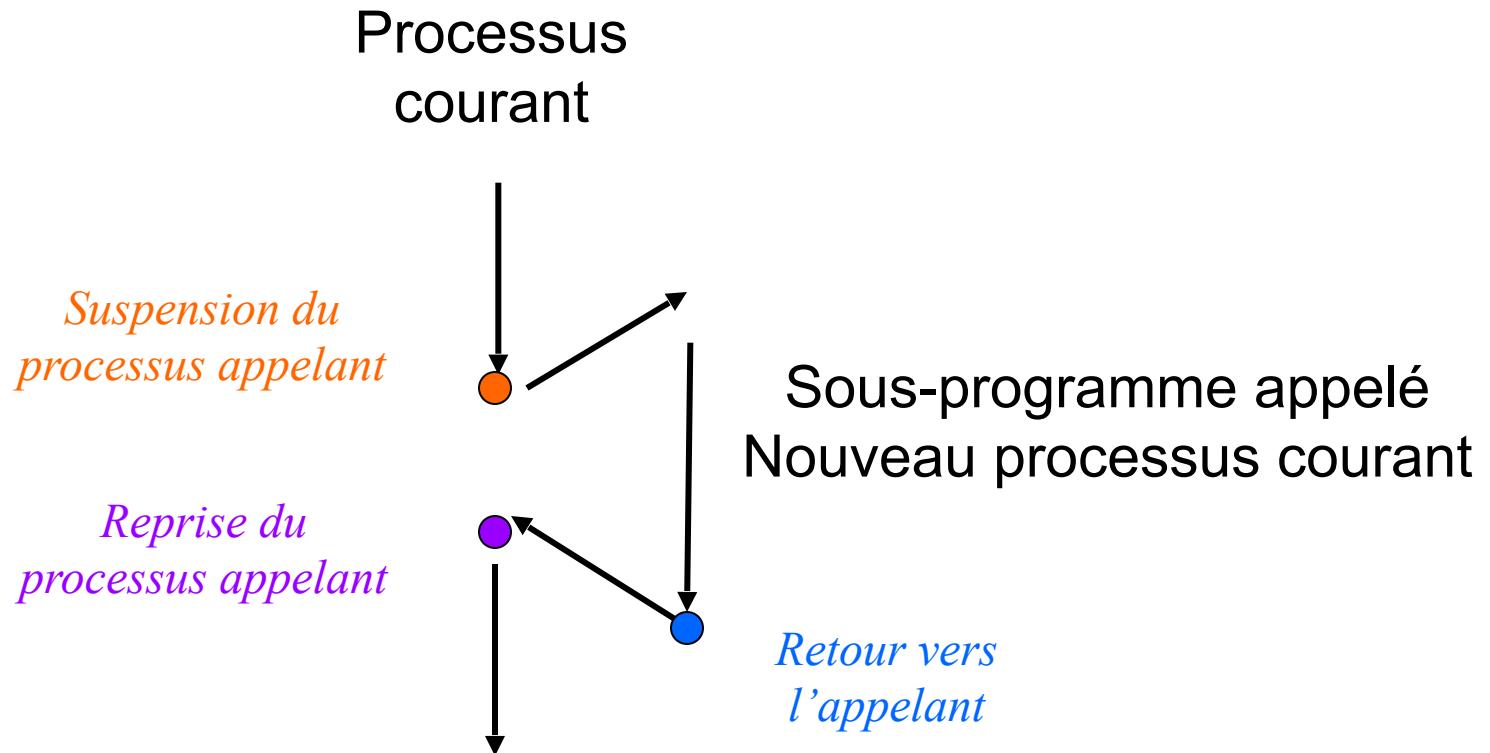
```
main(){  
    float X, Y;  
    printf("Entrez la valeur : ");  
    scanf("%f",&X);  
    Y = cos(X);  
    printf(" %f" ,Y);  
}
```

*appel du sous-programme d'affichage  
appel du sous-programme de lecture  
appel du sous-programme cos  
appel du sous-programme d'affichage*

## 2.6 Les sous-programmes-Fonctionnement

- Dans l'exemple précédent, le programme principal, lance successivement un ensemble de sous-programmes, chacun d'eux pouvant d'ailleurs lancer ses propres sous-programmes.
- Cela revient à suspendre l'exécution du programme en cours, autrement dit le processus du programme principal et à lancer le processus associé au sous-programme.

## 2.6 Les sous-programmes-Fonctionnement



## 2.6 Les sous-programmes-Intérêts

- Diviser pour mieux régner
- Méthode modulaire de conception utilisant le découpage d'un problème en sous-problèmes distincts
- Permet de réutiliser des programmes (sous-programmes) déjà développés et surtout validés

## 2.6 Les sous-programmes-Problématique de l'usage des sous-programmes

- Quel est le rôle du sous-programme qui est exécuté le premier ?
- Peut-on définir des variables communes à plusieurs sous-programmes et comment sont-elles définies et/ou modifiées ?
- Comment peut-on transmettre des informations spécifiques à un sous-programme lors de son lancement ?
- Comment un sous-programme peut renvoyer des informations au programme appelant ?

## 2.6 Les sous-programmes- Environnement d'un sous-programme

- L'environnement d'un sous-programme est l'ensemble des variables accessibles et des valeurs disponibles dans ce sous-programme, en particulier celles issues du programme appelant.
- Trois sortes de variables sont accessibles dans le sous-programme :
  - Les variables dites globales
  - Les variables dites locales
  - Les paramètres formels

## 2.6 Les sous-programmes-Environnement d'un sous-programme

- Les variables dites globales :
  - Celles définies dans le programme appelant et considérées comme disponibles dans le sous-programme
  - Elles peuvent donc être référencées partout dans le programme appelant et dans le sous-programme appelé
  - Leur portée est globale

## 2.6 Les sous-programmes-Environnement d'un sous-programme

- Les variables dites locales
  - Celles définie dans le corps du sous-programme, utiles pour son exécution propre et accessibles seulement dans ce sous-programme
  - Elles sont donc invisibles pour le programme appelant
  - Leur portée est locale

## 2.6 Les sous-programmes-Environnement d'un sous-programme

- Les paramètres formels
  - Les variables identifiées dans le sous-programme qui servent à l'échange d'information entre les programmes appelant et appelé
  - Leur portée est également locale.

## 2.6 Les sous-programmes-Les procédures

- Il s'agit de sous-programmes *nommés* qui représentent une ou plusieurs actions, et qui peuvent calculer et retourner une ou plusieurs valeurs.
- Une procédure « P » doit être définie une seule fois dans un algorithme « A » et peut être appelée plusieurs fois par « A » ou par une autre procédure de « A ».

## 2.6 Les sous-programmes-Les procédures

- La définition d'une procédure comprend trois parties:
- Partie 1 : L'entête  
**Procédure** *Nom\_procedure* (*Liste de paramètres formels*);
- Partie 2 : Déclaration des variables locales  
*type\_variable\_i Nom\_variable\_i*
- Partie 3 : Corps de la procédure  
**Début**  
*Instructions*  
**Fin**

## 2.6 Les sous-programmes-Les procédures

- **Exemple :**
  - Calcul de la somme de deux matrices carrées A et B.
  - Pour calculer la somme de deux matrices il faut lire d'abord ces deux matrices.
  - Ainsi, au lieu d'écrire deux fois le sous-programme de lecture d'une matrice, il est possible d'écrire plutôt une procédure « Saisir » et de l'utiliser pour saisir A et B.

## 2.6 Les sous-programmes-Les procédures (Paramètres formels & paramètres effectifs)

- Dans la déclaration de la procédure « Saisir », Les paramètres « X » et « dim » sont appelés paramètres **formels** dans la mesure où ils ne possèdent pas encore de valeurs.
- Lors de l'utilisation de la procédure dans l'algorithme principal, « A », « B » et « n » sont des paramètres **effectifs** (ou réels).
- **Remarque importante:** Dans une procédure, le nombre de paramètres formels est exactement égal au nombre paramètres effectifs.
- De même à chaque paramètre formel doit correspondre un paramètre effectif de même type.

## 2.6 Les sous-programmes-Les procédures (Passage des paramètres)

- Lors de la définition d'une procédure il est possible de choisir entre trois modes de transmission de paramètres:
  - **Mode donnée** (*En C on parle de Passage par valeur*)
  - **Mode Résultat** (*En C on parle de Passage par adresse*).
  - **Mode Donnée/Résultat** (*En C on parle de Passage par adresse*):

## 2.6 Les sous-programmes-Les procédures (Passage des paramètres-Mode données)

- Lors de l'appel d'une procédure un emplacement mémoire est réservé pour chaque paramètre formel.
- De même, un emplacement mémoire est aussi réservé pour chaque paramètre effectif lors de sa déclaration.
- Cependant, lors de l'appel de la procédure, les valeurs des paramètres effectifs sont copiées dans les paramètres formels.

## 2.6 Les sous-programmes-Les procédures (Passage des paramètres-Mode données)

- L'exécution des instructions de la procédure se fait avec les valeurs des paramètres formels et toute modification de ces derniers ne peut affecter en aucun cas celles des paramètres effectifs.
- Dans ce type de passage de paramètres, les valeurs des paramètres effectifs sont connues avant le début de l'exécution de la procédure et jouent le rôle uniquement d'entrées de la procédure.

## 2.6 Les sous-programmes-Les procédures (Passage des paramètres-Mode données)

**Procédure** carrés(**Données** X,Y :**entiers**)

**Entiers** A,B

**Début**

A  $\leftarrow$  X

B  $\leftarrow$  Y

A  $\leftarrow$  A\*A

B  $\leftarrow$  B\*B

**Fin**

**Entiers** M1,M2

**Début**

**Lire** M1, M2

carrés(M1,M2)

**écrire** « M1= »,M1, « M2= »,M2

**Fin**

## 2.6 Les sous-programmes-Les procédures (Passage des paramètres-Mode données)

**Procédure** carrés\_bis(**Données** X,Y :entiers)

**Début**

X ← X\*X

Entiers M1,M2

Y ← Y\*Y

**Début**

**écrire** « X= »,X, « Y= »,Y

Lire M1, M2

**Fin**

carrés\_bis(M1,M2)

**écrire** « M1= »,M1, « M2= »,M2

**Fin**

## 2.6 Les sous-programmes-Les procédures (Passage des paramètres-Mode résultats)

- Passage par adresse, par variable
- La différence principale entre le passage par valeur et le passage par adresse c'est que dans ce dernier **un seul emplacement mémoire** est réservé pour le paramètre formel et le paramètre effectif correspondant.
- Dans ce cas chaque paramètre formel de la procédure utilise **directement** l'emplacement mémoire du paramètre effectif correspondant.
- Toute modification du paramètre formel entraîne la même modification du paramètre effectif correspondant.

## 2.6 Les sous-programmes-Les procédures (Passage des paramètres-Mode résultats)

- Dans ce mode de passage de paramètres, les valeurs des paramètres effectifs sont inconnues au début de l'exécution de la procédure.
- Un paramètre formel utilisant ce type de passage ne peut être que le résultat de la procédure. D'où le nom du mode « Résultat ».
- Pour spécifier dans une procédure qu'il s'agit du mode « Résultat », il suffit d'ajouter dans l'algorithme la mention « **Résultat** » avant le paramètre formel

## 2.6 Les sous-programmes-Les procédures (Passage des paramètres-Mode résultats)

**Procédure** Somme(**Données** D,F:entier, **Résultat** R:entier)

- Se traduit en C par :

*void Somme(int D, int F, int \*R);*

- Contrairement au passage par valeur, dans ce cas, à l'appel de la procédure les valeurs des paramètres effectifs ne sont pas copiées dans les paramètres formels.
- Ces derniers utilisent directement l'emplacement mémoire (ou l'adresse) des paramètres effectifs.

## 2.6 Les sous-programmes-Les procédures (Passage des paramètres-Mode Données/Résultats)

- Passage par adresse, par référence
- Les paramètres formels jouent simultanément le rôle de donnée (i.e. d'entrée) et de résultat (sortie) de la procédure: d'où l'appellation « Donnée/Résultat ».
- A l'appel de la procédure, les valeurs des paramètres effectifs sont connues au début de l'exécution de la procédure mais elles seront modifiées à la fin de cette exécution: d'ou la seconde appellation « Donnée Modifiée ».

## 2.6 Les sous-programmes-Les procédures (Passage des paramètres-Mode Données/Résultats)

- Ces deux derniers modes (i.e. *Résultat* et *Donnée Modifiée*) se distinguent uniquement au niveau de l'algorithme.
- En C ces deux derniers modes sont exactement identiques : il s'agit dans les deux cas de *passage par adresse*.

## 2.6 Les sous-programmes-Les procédures (Passage des paramètres-Mode Données/Résultats)

- Pour spécifier dans une procédure qu'il s'agit du mode Donnée/Résultat (ou donnée modifiée), il suffit d'ajouter dans l'algorithme la mention
  - « **Donnée/Résultat** »
  - ou « **Donnée Modifiée** »devant le paramètre formel
- Dans le programme C le paramètre formel doit être un pointeur

## 2.6 Les sous-programmes-Les procédures (Passage des paramètres-Mode Données/Résultats)

- **Exemple :**

- Algorithme d'une procédure permettant de remplacer par 0 toutes les valeurs inférieures à 10 des matrices.
- La matrice qu'il faut fournir à cette procédure doit être une donnée ( i.e. qui contient déjà des valeurs).
- Le résultat de cette procédure c'est cette même matrice modifiée.
- Ainsi, dans ce cas le mode que l'on doit utiliser est « **Donnée/Résultat** » (ou **Donnée modifiée**).

## 2.6 Les sous-programmes-Les procédures (Passage des paramètres-Mode Données/Résultats)

**Procédure** Remplacer (**Donnée Modifiée** M:Matrices; **Donnée** dim: entier);  
entiers i, j

**Début**

**Pour** i $\leftarrow$  1 à dim **faire**

**Pour** j $\leftarrow$  1 à dim **faire**

**Si** M[i, j] < 10 **alors**

M[i, j]  $\leftarrow$  0

**Fsi**

**Fpour**

**Fpour**

**Fin;**

## 2.6 Les sous-programmes-Les fonctions

- Une fonction est un sous-programme qui calcule, à partir de paramètres éventuels, une valeur d'un certain type utilisable dans une expression du programme appelant.
- Les langages de programmation proposent de nombreuses fonctions prédéfinies : partie entière d'un nombre, racine carrée, fonctions trigonométriques, etc.

## 2.6 Les sous-programmes-Les fonctions

- Les fonctions sont comparables aux procédures à deux exceptions près :
  - Leur appel ne constitue pas à lui seul une instruction, mais figure dans une expression ;
  - Leur exécution produit un résultat qui prend la place de la fonction lors de l'évaluation de l'expression.

## 2.6 Les sous-programmes – Les fonctions

- La définition d'une fonction comprend aussi trois parties:

- Partie 1 : L'entête

**Fonction** *Nom\_fonction(Liste de paramètres formels)*: **Type\_fonction**

- Partie 2 : Déclaration des variables locales

*type\_variable\_i Nom\_variable\_i*

- Partie 3 : Corps de la fonction

**Début**

*Instructions*

**Retour** **valeur retournée**

**Fin**

## 2.6 Les sous-programmes – Les fonctions

- Il s'agit donc de sous-programmes *nommés* et *typés* qui calculent et retournent une et une seule valeur.
- Le type de cette valeur est celui de la fonction.



## 2.6 Les sous-programmes – Les fonctions (Passage des paramètres)

- Lors de la définition d'une fonction il ne doit, normalement, être utilisé qu'un seul mode de transmission de paramètres :
  - **Mode donnée** (*En C on parle de Passage par valeur*)
- *Cependant les langages de programmation permettent d'utiliser les autres modes.*

## 2.6 Les sous-programmes – Les fonctions (exemple)

- Fonction permettant de calculer  $e^x$  pour x donnée avec une précision  $10^{-3}$  et sachant que :

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \dots + \frac{x^n}{n!}$$

## 2.6 Les sous-programmes – Les fonctions (exemple)

Constante utilisée

$E=0.001$

Variables

Réel x

fonction expo(Donnée x: réel): réel

Entier i

réel f, ex

Début

$f \leftarrow 1$     $ex \leftarrow 1$     $i \leftarrow 1$

Tant que  $f \geq E$  faire

$f \leftarrow f * (x / i)$

$ex \leftarrow ex + f$

$i \leftarrow i + 1$

ftq

Retour ex;

Fin

## 2.6 Les sous-programmes – Les fonctions (exemple)

### Algorithme principal

Début

Écrire « Donner la valeur de x: »

Lire x

Écrire « L'exponentielle de x= », x, « est : », expo(x)

Fin

## 2.6 Les sous-programmes – Les fonctions (exemple - traduction en C)

```
#include <stdio.h>
#define E 0.001
float expo(float x) {
    int i;  float f,ex;
    f=1;  ex=1;  i=1;
    while (f>=E) {
        f=f*(x/i);
        ex=ex+f;
        i++;
    }
    return ex;
}
```

## 2.6 Les sous-programmes – Les fonctions (exemple - traduction en C)

```
/*Programme principal */  
main() {  
    float x;  
    printf("Donner la valeur de x : ");  
    scanf("%f",&x);  
    printf("L'exponentiel de x=%f est %f\n",x,expo(x));  
}
```

## 2.6 Les sous-programmes – Les fonctions (Fonctions récursives)

- Une fonction récursive possède donc la propriété de s'appeler elle-même dans sa définition
- Permet de coder des fonctions définies à partir de relations de récurrence

## 2.6 Les sous-programmes – Les fonctions (Fonctions récursives)

- Exemple calcul du factoriel d'un nombre entier :
  - 1<sup>ère</sup> définition :
    - $n! = 1 * 2 * 3 * 4 * \dots * n$
    - Traduction par une boucle :
  - 2<sup>ème</sup> définition :
    - $n! = \begin{cases} 1 & \text{si } n=0 \\ n * (n-1)! & \text{autre cas} \end{cases}$

## 2.6 Les sous-programmes – Les fonctions (Fonctions récursives)

**fonction** factoriel(**Donnée** n: entier): entier

**Entier** fac

**Début**

si n = 0 **alors**

    fac  $\leftarrow$  1

**sinon**

    fac  $\leftarrow$  n \* factoriel(n-1)

**fsi**

**Retour** fac;

**Fin**

## 2.7 – Les structures

- Les tableaux sont des paquets de données de même type.
- Les structures sont des ensembles de données non homogènes.
- Les données peuvent avoir des types différents.
- Les structures sont déclarées ou définies selon le modèle :

## 2.7 – Les structures

- Les structures sont déclarées ou définies selon le modèle :

```
struct nom_de_structure_facultatif {  
    la liste des données contenues dans la  
    structure  
} la liste des variables construites selon ce  
modèle.
```

## 2.7 – Les structures

- Exemple :

```
Struct etd {  
    char nom[20];  
    char prenom[20];  
    float note_info;  
    float note_phi;  
    float moyenne;  
} anriolle;
```

## 2.7 – Les structures

- etd est un nom de modèle de structure
- Anicet est un objet de type struct etd.
- Les différentes parties de la structure Anicet sont accessibles par :
  - Anicet.nom
  - Anicet.prenom
  - Anicet.note\_info
  - Anicet.note\_phi
  - Anicet.moyenne

# Fin