Contenu du module

- Introduction
- Types de base, variables, constantes
- Opérateurs et expressions
- Les entrées-sorties en C
- Les structures de contrôle
- Les tableaux
- Les pointeurs
- Des séries de TD et TP seront proposées pour mettre en pratique les connaissances acquises.

1.1 Introduction générale:

Un algorithme est une suite d'actions précises qui doivent être exécutées dans un ordre déterminé en vue de la résolution d'un problème donné.

Un programme informatique est une suite d'instructions écrites dans un langage compréhensible par l'ordinateur afin de résoudre un problème donné. Autrement dit, il s'agit d'une traduction d'un algorithme.

Pour que l'ordinateur puisse comprendre l'algorithme qu'on veut exécuter on utilise un langage intermédiaire: compréhensible par l'homme et qui sera ensuite transformé en langage machine (que des 0 et des 1) pour être exploitable par le processeur.

Comme les langages naturels, tous les langages de programmation ont des règles à respecter.

1.2 Introduction au langage C

Le langage C a été créé en 1972 par Denis Ritchie afin de développer un système d'exploitation (UNIX). Mais ses qualités opérationnelles l'ont vite fait adopter par une large communauté de programmeurs.

Langage normalisé par l'ANSI (American National Standards Institute), puis par l'ISO (International Standards Organization) en 1990 et en 1993 par le CEN (comité européen de normalisation) → « C ANSI » ou « C norme ANSI ».

Le langage C a donné naissance à de nombreux langages dérivés comme le C++ qui sera étudié dans le Semestre 5-SMA. Le langage C sera utilisé en SMA-S4 pour implémenter les structures de données

2. Présentation par l'exemple:

Dans cette exemple, nous allons découvrir comment s'expriment les instructions de base (déclaration, affectation, lecture et écriture) ainsi que la structure sélective.

2.1 Exemple d'algorithme de calcul de la racine carrée

```
Algorithme calcul_racine_carree;
Variables:
    x, racx: réels
Début
Ecrire ("Bonjour, je vais vous calculer la racine carree") ;
Ecrire ("Donnez un nombre : ") ;
Lire(x) ;
Si (x < 0.0) alors
                Ecrire  ("Le nombre  " ,x,  " ne possede pas de racine carree") ;
Sinon
        racx ← x^0.5;
        Ecrire ("Le nombre " , x , " a pour racine carree :", racx) ;
FinSi
Ecrire  ("Travail termine - Au revoir") ;
Fin
```

2.1 Traduction de l'algorithme précédent en un programme en langage C :

```
#include <stdio.h>
#include <math.h>
#include <conio.h>
main()
{
float x ;
float racx ;
printf ("Bonjour, je vais vous calculer la racine carree") ;
printf ("Donnez un nombre : ") ;
scanf ("%f", &x) ;
if (x < 0.0)
printf ("Le nombre %f ne possede pas de racine carree\n", x) ;
else
{ racx = sqrt (x) ;
printf ("Le nombre %f a pour racine carree : %f\n", x, racx) ;
}
printf ("Travail termine - Au revoir") ;
getch();
}
```

Résultats :

Bonjour, je vais vous calculer la racine carree
Donnez un nombre : 4
Le nombre 4.000000 a pour racine carree : 2.000000
Travail termine - Au revoir

2.2 Structure d'un programme en langage C

La ligne : **main()**

• se nomme un "en-tête". Elle précise que ce qui sera décrit à sa suite est le "*programme principal" (main).* Le programme principal est délimité par les accolades "{" et "}".

• *Les* instructions situées entre ces accolades forment un "bloc".

2.3 Déclarations

Les trois instructions ci-dessous sont des "**déclarations**".

float x ; → x et racx de type float (destinée à contenir des nombres flottants)
float racx ;

Remarque: les déclarations des types des variables sont obligatoires et doivent être regroupées au début du programme.

2.4 Ecrire des informations : la fonction printf

L'instruction : **printf ("Bonjour, je vais vous calculer la racine carree") ;**
appelle une fonction "prédéfinie" (fournie avec le langage) nommée *printf.*
Ici, cette fonction reçoit un argument : **"Bonjour, je vais vous calculer la racine carree"**
Les guillemets servent à délimiter une "chaîne de caractères"

L'instruction : **printf ("Le nombre %f a pour racine carrée : %f\n", x, racx) ;**

Ici, printf reçoit deux arguments x et racx.
%f (f : code format), le **Format** précise comment afficher les informations fournies par les arguments.

2.5 Lire des informations : la fonction scanf

L'instruction : **scanf ("%f", &x) ;**
Est un appel de la fonction prédéfinie *scanf* dont le rôle est de lire une information au clavier.
Scanf possède en premier argument un format, ici: **"%f"**

La fonction *scanf* range la valeur lue dans l'emplacement correspondant à la variable x, c'est-à-dire à son **adresse.** L'opérateur & signifie "*adresse de".*

2.6 Faire des choix: l'instruction if

Les lignes:

```
if (x < 0.0)
      printf ("Le nombre %f ne possede pas de racine carree\n", x) ;
else
{ racx = sqrt (x) ;
printf ("Le nombre %f a pour racine carree : %f\n", x, racx) ;
}
```

Si la condition (x < 0.0) est **vraie**, on exécute l'instruction suivante:

```
printf ("Le nombre %f ne possede pas de racine carre\n", x) ;
```

Si la condition est **fausse**, on exécute l'instruction suivant le mot else:

```
{ racx = sqrt (x) ;
  printf ("Le nombre %f a pour racine carree : %f\n", x, racx) ;
}
```

Remarque: La fonction *sqrt de la bibliothèque <math.h>* fournit la valeur de la racine carrée d'une valeur flottante qu'on lui transmet en argument.

2.7 Les directives au préprocesseur

Les trois premières lignes :

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

Il s'agit de "directives" qui seront prises en compte avant la traduction (compilation) du programme.

Ces directives doivent être écrites à raison d'une par ligne et elles doivent obligatoirement commencer en début de ligne.

Les trois directives demandent d'introduire des instructions situées dans les fichiers *stdio.h et math.h et conio.h*

lors que vous faites appel à une fonction prédéfinie, il est nécessaire d'incorporer de tels fichiers, nommés "fichiers en-têtes", qui contiennent des déclarations appropriées concernant cette fonction : *stdio.h pour printf et scanf et math.h pour sqrt.*

3. Règles d'écriture

3.1 Les identificateurs

Les identificateurs servent à désigner les différents "objets" manipulés par le programme : variables, fonctions, etc. les identificateurs sont composés d'une suite de caractères alphanumériques. Le premier d'entre eux étant nécessairement une **lettre**.

Règles d 'écriture des identificateurs:

- un identificateur peut contenir des lettres minuscules ou majuscules, des chiffres, ou le caractère spécial de soulignement « _ ». Par contre, il ne doit pas commencer par un chiffre ou posséder des lettres accentuées.

- les espaces ne sont pas admis dans l'identificateur ;

- les majuscules sont distinguées des minuscules, ainsi : SMA et sma désignent deux variables différentes.

- un identificateur ne peut pas être un mot clé du langage

3.2 Les Mots-clés

Les mots-clés, sont réservés pour le langage lui-même et ne peuvent pas être utilisés comme identificateurs. L'ANSI C compte 32 mots-clés :

auto	const	double	float	int	short	struct	unsigned
break	continue	else	for	long	signed	switch	void
case	default	enum	goto	register	sizeof	typedef	volatile
char	do	extern	if	return	static	union	while

3.3 Les Séparateurs

Deux identificateurs successifs doivent être séparés par une virgule et toutes les instructions se terminent par un point-virgule .

Exemple: int n, somme, moyen;

3.4 Le format libre

le langage C autorise une "mise en page" parfaitement libre. Une instruction peut s'étendre sur un nombre quelconque de lignes, et une même ligne peut comporter autant d'instructions que voulu.

Exemple:

```
main(
) { int i ; float
x; float racx ;
 printf ("Bonjour\n") ;
```

il faut bien structurer les programmes pour faciliter la compréhension par autres personnes. (faciliter au correcteur de votre feuille d'examen en programmation les opérations de lecture et de compréhension !)

3.5 Les Commentaires

Il s'agit de textes explicatifs destinés aux lecteurs du programme et qui n'ont aucune incidence sur sa compilation. Un commentaire débute par /* et se termine par */.

Exemple:

```
1.  /* programme de calcul de racines carrées */
2.  /* commentaire s'étendant
    sur plusieurs lignes de programme source */
```

4. Création d'un programme en langage C

La création d'un programme comporte trois étapes: l'édition du programme, la compilation et l'édition des liens.

4.1 L'édition du programme

Consiste à créer, à partir du clavier, tout ou partie du texte d'un programme « programme source ». Le fichier source porte l'extension C.

4.2 La compilation

consiste à traduire le programme source en langage machine, en faisant appel à un programme nommé compilateur.

4.3 L'édition de liens

Le module objet créé par le compilateur n'est pas directement exécutable. Il lui manque les différents modules objet correspondant aux fonctions prédéfinies (printf, scanf, sqrt…).

L'éditeur de liens recherche dans la bibliothèque standard (collection de modules objet) les modules objet nécessaires.

Le résultat de l'édition de liens est un programme exécutable: un ensemble autonome d'instructions en langage machine.

1. Introduction

Le C est un langage *typé. Cela signifie que toute* ***variable****, constante ou* fonction est d'un type précis. Le type d'un objet définit la façon dont il est représenté en mémoire.
La mémoire de l'ordinateur est un ensemble de "positions binaires" nommées bits. Les bits sont regroupés en octets (8 bits), et chaque octet est repéré par son adresse.
Les types de base du langage C se répartissent en trois grandes catégories en fonction de la nature des informations qu'ils permettent de représenter : nombres entiers, nombres flottants et caractères.

2. Les types de base

2.1 Les types entiers

Le mot clé **int** correspond à la représentation de nombres entiers. Un bit est réservé pour le signe du nombre et les autres bits servent à représenter sa valeur absolue.
Les types d'entiers sont: **short int (short), int, long int (long)**
Le tableau suivant résume les caractéristiques des types entiers:

définition	*description*	Valeur *min*	Valeur *max*	*nombre d'octets*
short	entier court	-32768	32767	2
int	entier	-32768 -2 147 483 648	32767 2 147 483 647	2 (Processeur 16 bits) 4 (Processeur 32 bits)
long	entier long	-2147483648	2147483647	4

Remarque: la façon la plus naturelle d'introduire une constante entière dans un programme est de l'écrire sous forme **décimale**, avec ou sans signe. Par exemple: +245 88 -34

2.2 Les types flottants

Les types flottants permettent de représenter, de manière approchée, une partie des nombres réels. Les nombres à virgule flottante possèdent un signe **s** (dans {-1, 1}), une **mantisse** *m* et un **exposant** *e*. Un tel triplet représente un réel ***s.m.10e*** **.**

	<+|-> <*mantisse*> * $10^{<exposant>}$
<+|->	est le signe positif ou négatif du nombre
<*mantisse*>	est un décimal positif avec un seul chiffre devant la virgule.
<*exposant*>	est un entier relatif

Exemple: $2.5\ 10^{24}$, $0.245\ 10^{-6}$

Le tableau suivant résume les caractéristiques des types flottants:

définition	Valeur *min*	Valeur *max*	*octets*
float	$3.4 * 10^{-38}$	$3.4 * 10^{38}$	4
double	$1.7 * 10^{-308}$	$1.7 * 10^{308}$	8
long double	$3.4 * 10^{-4932}$	$1.1 * 10^{4932}$	10

Remarque:

Les constantes flottantes peuvent s'écrire indifféremment suivant l'une des deux notations:

- Décimale : doit comporter obligatoirement un point (partie entière.Partie décimale 16.75)
- Exponentielle : utilise la lettre **e** pour introduire un exposant entier (puissance de 10), avec ou sans signe.

Exemple: **42.5e-3** **0.0425** (les deux écritures sont équivalentes)

2.3 Le type caractère

C permet de manipuler des caractères codés en mémoire sur 1 octet. Une des particularités du type char en C est qu'il peut être assimilé à un entier. Par exemple, si c est de type char, l'expression c+1 est valide. Elle désigne le caractère suivant dans le code ASCII.

Remarques:

- Les constantes de type "caractère", lorsqu'elles correspondent à des caractères imprimables, se notent de façon classique, en écrivant ente apostrophes (quottes) le caractère. Exemple: 'a', 'f', '+'
- Une chaîne de caractères est un suite de caractères entourés par des guillemets. Par exemple: ''Ceci est une chaîne de caractères''
- Certains caractères non imprimables possèdent une représentation conventionnelle utilisant le caractère "\" . Voici des exemples de ces caractères.

NOTATION EN C	SIGNIFICATION
\a	cloche ou bip (alert ou audible bell)
\n	Saut de ligne (Line Feed)
\t	Tabulation horizontale (Horizontal Tab)

En résumé, il y a 4 types de base, les autres sont dérivés de ceux-ci.

Type	Signification	Exemples de valeur
char	Caractère unique	'a' 'A' 'z' 'Z' '\n' 'a' 'A' 'z' 'Z' '\n' Varie de –128 à 127
int	Nombre entier	0 1 -1 4589 32000
float	Nombre réel simple	0.0 1.0 3.14 5.32 -1.23
double	Nombre réel double précision	0.0 1.0E–10 1.0 -1.34567896

3. Initialisation et constantes

- La directive **#define** permet de donner une valeur à un symbole. Dans ce cas, le préprocesseur effectue le remplacement correspondant avant la compilation.
- Il est possible d'initialiser une variable lors de sa déclaration comme dans :

int n = 10;

- Il est possible de déclarer que la valeur d'une variable ne doit pas changer lors de l'exécution du programme. Par exemple :

Const int n = 20 ;

1. Introduction

Le langage C est l'un des langages les plus fournis en opérateurs. Cette richesse se manifeste au niveau des opérateurs classiques (*arithmétiques, relationnels, logiques), moins classiques (manipulations de bits) et les* opérateurs originaux *d'affectation et d'incrémentation.*

2. Le opérateurs

2.1 L'affectation

En C, l'affectation est symbolisée par le signe **=**. Sa syntaxe est la suivante :

variable = expression

Le terme de gauche de l'affectation peut être une variable simple, un élément de tableau mais pas une constante. Cette expression a pour effet d'évaluer expression et d'affecter la valeur obtenue à variable.

L'affectation effectue une ***conversion de type implicite*** *: la valeur de l'expression* est convertie dans le type du terme de gauche. Par exemple, le programme suivant:

```
#include<stdio.h>
#include<conio.h>
main()
{int i, j = 2;     float x = 2.5;
i = j + x;   x=i+6; printf("\n %f \n",x);
getch();
}
```

imprime pour x la valeur 10.0 (et non 10.5), car dans l'instruction i = j + x; l'expression x a été convertie en entier.

Remarque:

L'opérateur d'affectation possède une **associativité de droite à gauche**. Ce qui permet à une expression telle que: **i = j = 5**

d'affecter la valeur 5 à la variable j. la valeur finale de cette expression est affectée à i, càd 5.

2.2 Les opérateurs arithmétiques

2.2.1 Présentation des opérateurs

Les opérateurs arithmétiques classiques sont l'opérateur unaire - (changement de signe) ainsi que les opérateurs binaires (càd portant sur deux opérandes):

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

Ces opérateurs agissent de la façon attendue sur les entiers comme sur les flottants. Leurs seules spécificités sont les suivantes :

○ Le C ne dispose que de la notation / pour désigner à la fois la division entière et la division entre flottants. Si les deux opérandes sont de type entier, l'opérateur / produira une division entière (quotient de la division). Par contre, il délivrera une valeur flottante dès que l'un des opérandes est un flottant. Par exemple,

```
float x;
x = 3 / 2;
```

affecte à x la valeur 1. Par contre

```
x = 3 / 2.;
```

affecte à x la valeur 1.5.

○ L'opérateur % ne s'applique qu' à des opérandes de type entier.

Remarque:

Il n'y a pas en C d'opérateur effectuant l'élévation à la puissance. De façon générale, il faut utiliser la fonction pow(x,y) de la librairie math.h pour calculer $\mathbf{x^y}$.

2.2.2 Les priorités des opérateurs arithmétiques

- Les opérateurs unaires + et - ont la priorité la plus élevée. On trouve ensuite, à un même niveau, les opérateurs *, / et %. Enfin, sur un dernier niveau, apparaissent les opérateurs binaires + et -.
- En cas de priorités identiques, les calculs s'effectuent de "gauche à droite". On dit que l'on a affaire à une "*associativité de gauche à droite*".
- Les parenthèses permettent d'outrepasser ces règles de priorité, en forçant le calcul préalable de l'expression qu'elles contiennent.

Exemples:

a + b * c	a + (b * c)
a * b + c % d	(a * b) + (c % d)
- c % d	(- c) % d
- a + c % d	(- a) + (c % d)

2.2.3 La conversion implicite des types

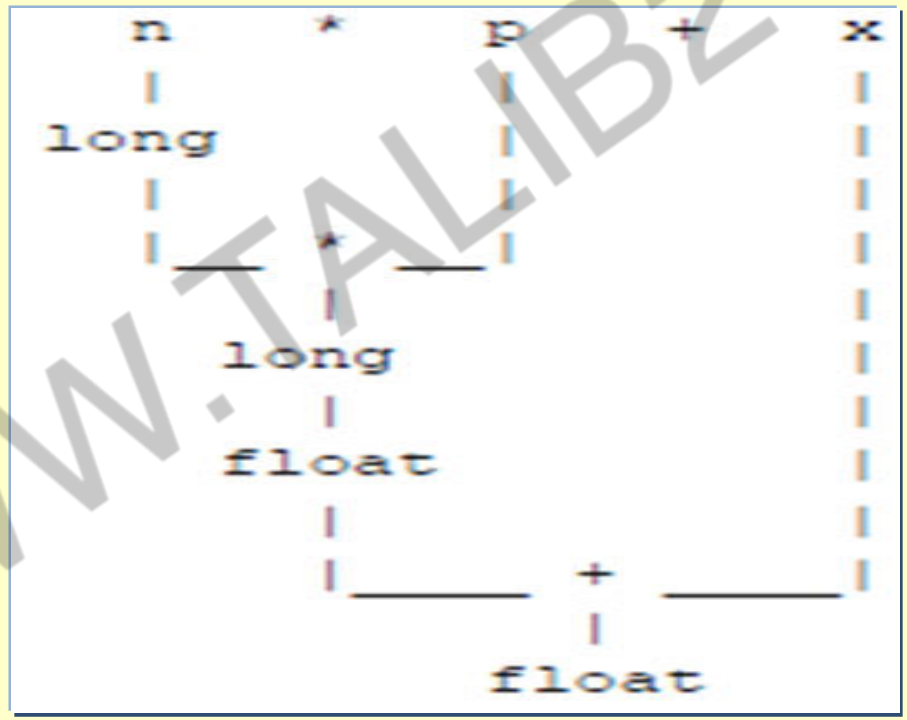
une conversion implicite consiste en une modification du type de donnée effectuée automatiquement par le compilateur. Cela signifie que lorsque l'on va stocker un type de donnée dans une variable déclarée avec un autre type, le compilateur ne retournera pas d'erreur mais effectuera une conversion *implicite* de la donnée avant de l'affecter à la variable. (voir l'exemple d'affectation dans la diapositive 19).

❑ Les conversions d'ajustement de type

Une conversion telle que **int -> float** se nomme une conversion d'ajustement de type. Une telle conversion ne peut se faire que suivant une "hiérarchie" qui permet de ne pas dénaturer la valeur initiale, à savoir : **int -> long -> float -> double ->long double**

Exemple:

Si n est de type int, p de type long et x de type float,
l'expression : n * p + x est évaluée suivant ce schéma :

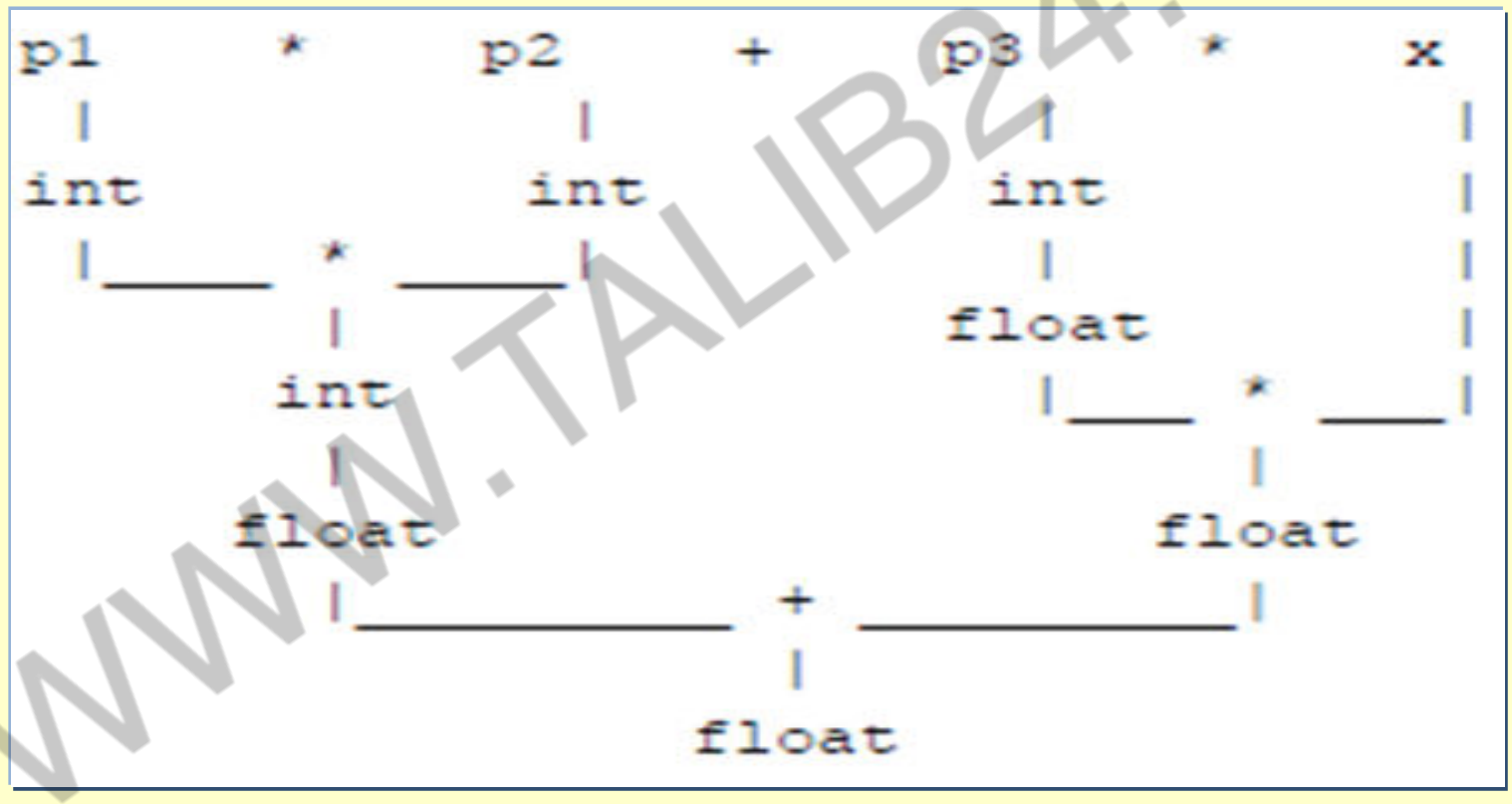


❑Les conversions systématiques “ Promotions numériques”

le langage C prévoit que toute valeur de type **char** ou **short** apparaissant dans une expression est d'abord convertie en int, et cela sans considérer les types des autres opérandes.

Exemple:

Si p1, p2 et p3 sont de type short et x de type float, l’expression p1*p2+p3*x est évaluée comme l’indique le schéma:



2.3 Les opérateurs relationnels

>	strictement supérieur
>=	supérieur ou égal
<	strictement inférieur
<=	inférieur ou égal
==	égal
!=	différent

Leur syntaxe est

expression-1 op expression-2

Les deux expressions sont évaluées puis comparées. La valeur rendue est de type int (il n'y a pas de type booléen en C); elle vaut **1** si la condition est vraie, et **0** sinon.

Attention : ne pas confondre l'opérateur de test d'égalité == avec l'opérateur d'affection =.

2.4 Les opérateurs logiques

&& et logique

|| ou logique

! négation logique

la valeur retournée par ces opérateurs est un int qui vaut 1 si la condition est vraie et 0 sinon.

Dans une expression de type: ***expression-1 op-1 expression-2 op-2 ...expression-n***

l'évaluation se fait de gauche à droite et s'arrête dès que le résultat final est déterminé.

Remarque: l'opérateur || est moins prioritaire que &&.

2.7 Les opérateurs d'incrémentation et de décrémentation

Les opérateurs d'incrémentation **++** et de décrémentation **--** s'utilisent aussi bien en suffixe (i++) qu'en préfixe (++i).

Dans les deux cas la variable i sera incrémentée, toutefois dans la notation suffixe la valeur retournée sera l'ancienne valeur de i alors que dans la notation préfixe se sera la nouvelle. Par exemple,

```
int a = 3, b, c;
b = ++a; /* a et b valent 4 */
c = b++; /* c vaut 4 et b vaut 5 */
```

2.8 L'opérateur de conversion de type

L'opérateur de conversion de type, appelé **cast**, permet de modifier **explicitement** le type d'un objet. On écrit ***(type) objet***

Par exemple,

```
main()
{
int i = 3, j = 2;
printf("%f \n",(float)i/j);
}
```

retourne la valeur 1.5.

2.9 L'opérateur adresse

L'opérateur d'adresse **&** appliqué à une variable retourne l'adresse-mémoire de cette variable. La syntaxe est **&objet**

1. Les fonctions d'entrées-sorties

Il s'agit des fonctions de la librairie standard stdio.h utilisées avec les unités classiques d'entrées-sorties, qui sont respectivement le clavier et l'écran. Sur certains compilateurs, l'appel à la librairie stdio.h par la directive au préprocesseur **#include <stdio.h>** n'est pas nécessaire pour utiliser printf et scanf.

2. La fonction printf

La fonction **printf** est une fonction d'impression formatée (affichage sur l'écran), ce qui signifie que les données sont converties selon le format particulier choisi. Sa syntaxe est

printf(*" chaîne de contrôle ",expression-1, ..., expression-n);*

La **chaîne de contrôle** contient le texte à afficher et **les spécifications de format** correspondant à chaque **expression** de la liste.

Les spécifications de format sont introduites par le caractère **%**, suivi d'un caractère désignant **le format** d'impression (code de conversion).

2.1 Les principaux formats*

c **char** : caractère affiché "en clair"

d **int**

u unsigned int

ld long

lu unsigned long

f **double ou float** écrit en notation "décimale" avec six chiffres après le point (123.456789)

e double ou float écrit en notation "exponentielle" avec six chiffres après le point décimal

s **chaîne de caractères**

3. La fonction scanf

La fonction **scanf** permet de faire une lecture formatée du flux standard d'entrée (le clavier par défaut). Sa syntaxe est **scanf("<format>",<AdrVar1>,<AdrVar2>, ...)**

- "<format>" : format de lecture des données
- <AdrVar1>,... : adresses des variables auxquelles les données seront attribuées

La chaîne de format détermine comment les données reçues doivent être interprétées.
Les données reçues correctement sont mémorisées successivement aux **adresses** indiquées par **<AdrVar1>,...** .
L'adresse d'une variable est indiquée par le nom de la variable précédé du signe **&**.

3.1 Les principaux formats

c	char
d	int
u	unsigned int
hd	short int
hu	unsigned short
ld	long int
lu	unsigned long
f ou e	float (notation décimale ou exponentielle)
lf ou le	double
s	chaîne de caractères

1. Introduction

Dans un programme, les instructions sont exécutées séquentiellement, càd dans l'ordre où elles apparaissent. Or la puissance et le "comportement intelligent" d'un programme proviennent essentiellement :

- de la possibilité d'effectuer des "**choix",** de se comporter différemment suivant les circonstances.
- de la possibilité d'effectuer des "**itérations**" (**boucles)**, autrement dit de répéter plusieurs fois un ensemble donné d'instructions.

2. Branchement conditionnel

2.1 l'instruction if

La syntaxe de l'instruction if

```
if ( expression_1 )
instruction_1
else if ( expression_2 )
instruction_2
.....
else
instruction_n
```

nombre quelconque de else if (….)

expression_1, expression_2: expressions quelconques

instruction_1, _2,… et instruction_n : instructions quelconques :

- simple (terminée par un point virgule),
- bloc (placé entre { }),
- instruction structurée (choix, boucle).

Si l'expression_1 est vraie on exécute l'instruction_1.
Sinon, si l'expression_2 est vraie on exécute l'instruction_2.
…… …….
Sinon (else), on exécute l'instruction_n .

2.2 Imbrication des instructions if

L'instruction d'un if peut contenir un autre if

if (expression)
instruction contenant d'autres instructions if

Remarque: un else se rapporte toujours au dernier if rencontré auquel un else n'a pas encore été attribué.

Exercice 5.1 :*

Ecrivez un programme de facturation qui lit en donnée un simple prix hors taxes et calcule le prix TTC correspondant (avec un taux de TVA constant de 20%). Il établit ensuite une remise dont le taux dépend de la valeur ainsi obtenue, à savoir :

* 0 % pour un montant inférieur à 1 000 dh
* 1 % pour un montant supérieur ou égal à 1 000 dh et inférieur à 2 000 dh
* 3 % pour un montant supérieur ou égal à 2 000 dh et inférieur à 5 000 dh
* 5 % pour un montant supérieur ou égal à 5 000 dh

2.3 l'instruction switch

La syntaxe de l'instruction switch

```
switch ( expression )
{
case constante_1:
liste d'instructions_1
break;
case constante_2:
Liste d'instructions_2
break;  ← provoque une sortie du bloc
...
default:
liste d'instructions_n
break;
}
```

Si la valeur de ***expression*** est égale à l'une des ***constantes***, la liste d'instructions correspondant est exécutée. Sinon la liste d'instructions_n correspondant à default est exécutée.

Exemple :*

Écrivez un programme qui lit un chiffre en entrée et l'affiche en lettre sur l'écran.

0 -> Nul, 1 -> un et 2 -> deux.

```
#include <stdio.h>
#include<conio.h>
main()
{char c;
float a,b,res;
printf("donner a et b \n ");
scanf("%f%f",&a,&b);
printf("donner l operation c:");
fflush(stdin);
scanf("%c",&c);
switch(c)
{case '+':res=a+b;
                break;
 case '-':res=a-b;
                break;
 case '*':res=a*b;
                break;
 case '/':res=a/b;
                break;
 default : printf("\n c'est pas une operation");
}
printf("resultat : %f",res);
getch();
}
```

3. l'Itération (les boucles)

Les boucles permettent de répéter une série d'instructions tant qu'une condition est vérifiée.
Le langage C fournit trois instructions pour réaliser des boucles: for, while et do..while.

3.1 la boucle déterministe – la boucle for

Les boucles déterministes sont des boucles dont le nombre d'itérations est connu.

La syntaxe de **for** est:

```
for ( expression 1 ; expression 2 ; expression 3) {
instructions;
}
```

expression 1: une instruction d'initialisation de la variable de contrôle **« i** par exemple **»**
expression 2: une condition logique qui teste la val&eur de la variable de contrôle
expression 3: une instruction qui fait évoluer la valeur de la variable de contrôle (incrémentation)

Exemple 5.1:

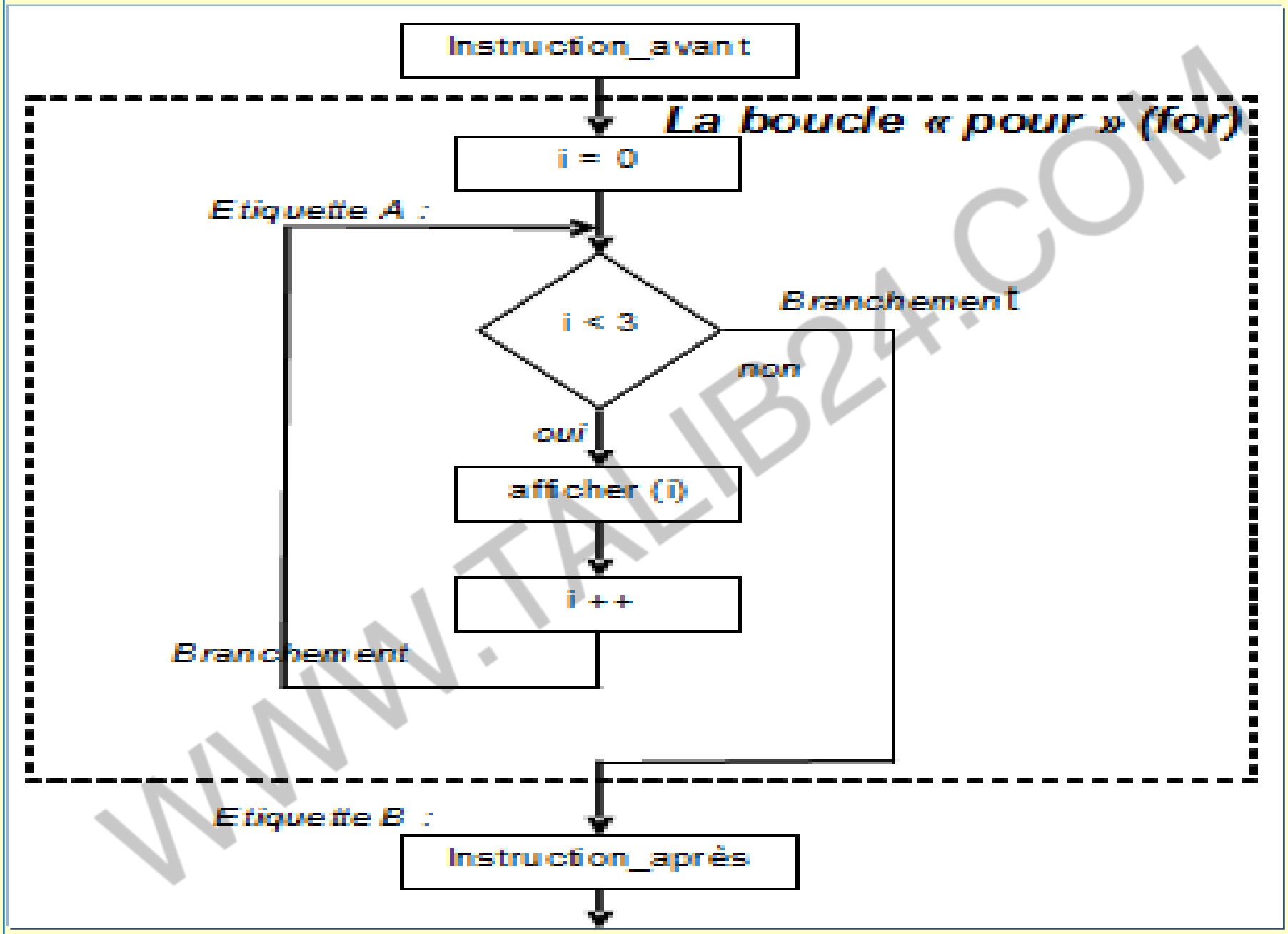
```
main()
{
int i ;
for ( i=1 ; i<=5 ; i++ )
{ printf ("bonjour ") ;
printf ("%d fois\n", i) ;
}}
```

```
Bonjour 1 fois
Bonjour 2 fois
Bonjour 3 fois
Bonjour 4 fois
Bonjour 5 fois
```

Exemple 5.2:

```
1   #include <stdio.h>
2   #include <conio.h>
3   main()
4   {int i;
5   printf("Avant la boucle\n");
6
7   for(i=0;i<3;i++)
8   {printf("i=%d\n",i);}
9
10  printf("Apres la boucle");
11
12  }
```

```
Avant la boucle
i=0
i=1
i=2
Apres la boucle
--------------------
```



3.2 la boucle indéterministe

Lorsque le nombre d'itérations qu'effectuera la boucle n'est pas connu d'avance ou que le pas est variable, il existe deux autres types de boucles, à savoir while et do---while.

3.2.1 la boucle while

La syntaxe de **While** est:

```
while ( expression) {
instructions;
}
```

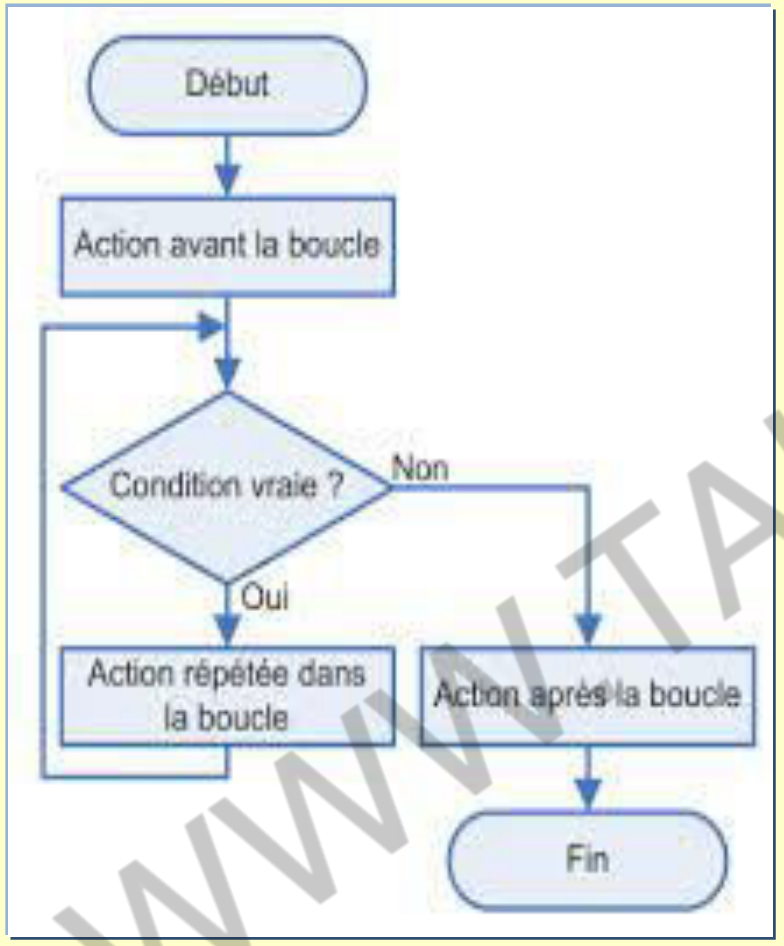
Tant que expression est vérifiée (i.e., non nulle), instruction est exécutée. Si expression est nulle au départ, instruction **ne sera jamais exécutée**.

Par exemple, le programme suivant calcule la somme de plusieurs nombres

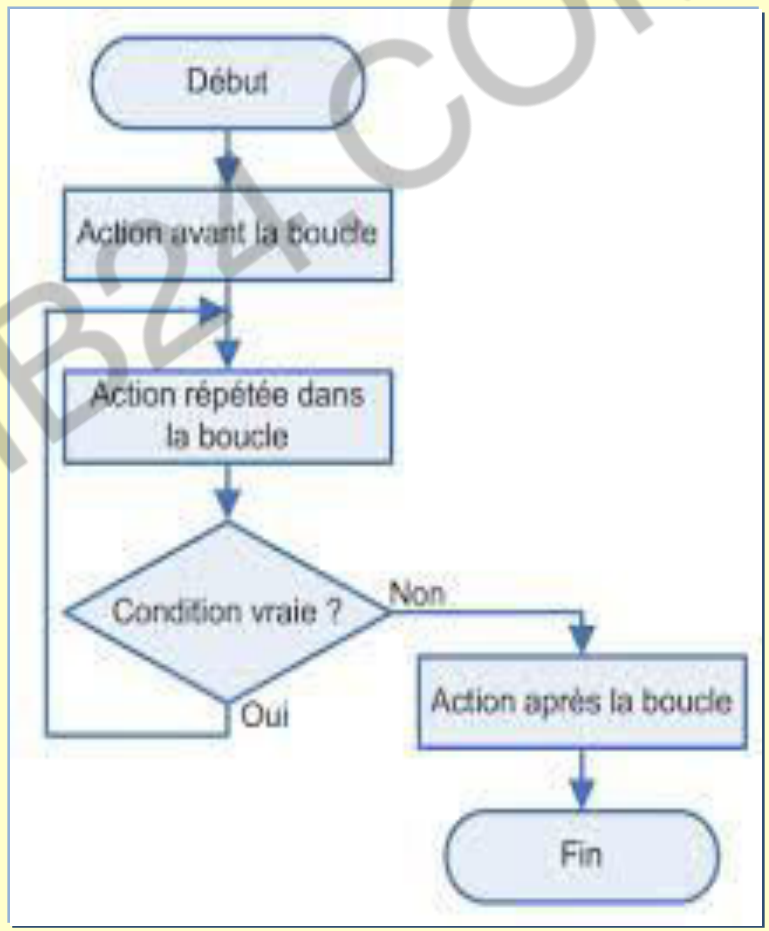
```
main()
{
int n, som ;
som = 0 ;
while (som<100)
{ printf ("donnez un nombre : ") ;
scanf ("%d", &n) ;
som += n ;
}
printf ("somme obtenue : %d", som) ;}
```

```
donnez un nombre : 15
donnez un nombre : 25
donnez un nombre : 12
donnez un nombre : 60
somme obtenue : 112
```

Organigramme de
la boucle while



Organigramme de
la boucle do…while



Exemple 5.1 – utilisation de la boucle while

```
main()
{
int i = 0 ;
while (i < 5 )
{
i++;
printf ("bonjour ") ;
printf ("%d fois\n", i) ;
}
```

```
Bonjour 1 fois
Bonjour 2 fois
Bonjour 3 fois
Bonjour 4 fois
Bonjour 5 fois
```

3.2.2 la boucle do --- while

do
{
Instructions
}
while (*expression);*

Ici, instructions seront exécutées tant que expression est vrai. Cela signifie donc que instructions sont toujours exécutées au moins une fois.

Exemple 5.2 :

Un programme qui demande à nombre à l'utilisateur (en affichant la valeur lue) tant qu'il ne fournit pas une valeur positive.

```
main()
{
int n ;
do
{ printf ("donnez un nb >0 : ") ;
scanf ("%d", &n) ;
printf ("vous avez fourni %d\n", n) ;
}
while (n<=0) ;
printf ("réponse correcte") ;
}
```

```
donnez un nb >0 : -3
vous avez fourni -3
donnez un nb >0 : -9
vous avez fourni -9
donnez un nb >0 : 12
vous avez fourni 12
réponse correcte
```

4. Les instructions de branchement non conditionnel

4.1 Branchement non conditionnel *break*

L'instruction break permet d'interrompre le déroulement de la boucle, et passe à la première instruction qui suit la boucle.

Exemple 5.3 :

```
main()
{
int i;
for (i = 0; i < 5; i++)
{
printf("i = %d\n",i);
if (i == 3)
break;
}
printf("valeur de i a la sortie de la boucle = %d\n",i);
}
```

imprime à l'écran

```
i = 0
i = 1
i = 2
i = 3
valeur de i a la sortie de la boucle = 3
```

4.2 Branchement non conditionnel *continue*

L'instruction continue permet de passer directement au tour de boucle suivant, sans exécuter les autres instructions de la boucle.

```
main()
{ int i;
for (i = 0; i < 5; i++)
{ if (i == 3)
continue;
printf("i = %d\n",i);
}
printf("valeur de i a la sortie de la boucle = %d\n",i);
}
```

imprime à l'écran

```
i = 0
i = 1
i = 2
i = 4
valeur de i a la sortie de la boucle = 4
```

4.3 Branchement non conditionnel *goto*

L'instruction goto permet d'effectuer un saut jusqu'à l'instruction *etiquette correspondant.*

```
main()
{
int i;
for (i = 0; i < 5; i++)
{
printf("i = %d\n",i);
if (i == 3)
goto sortie;
}
sortie : printf("valeur de i a la sortie de la boucle = %d\n",i);
}
```

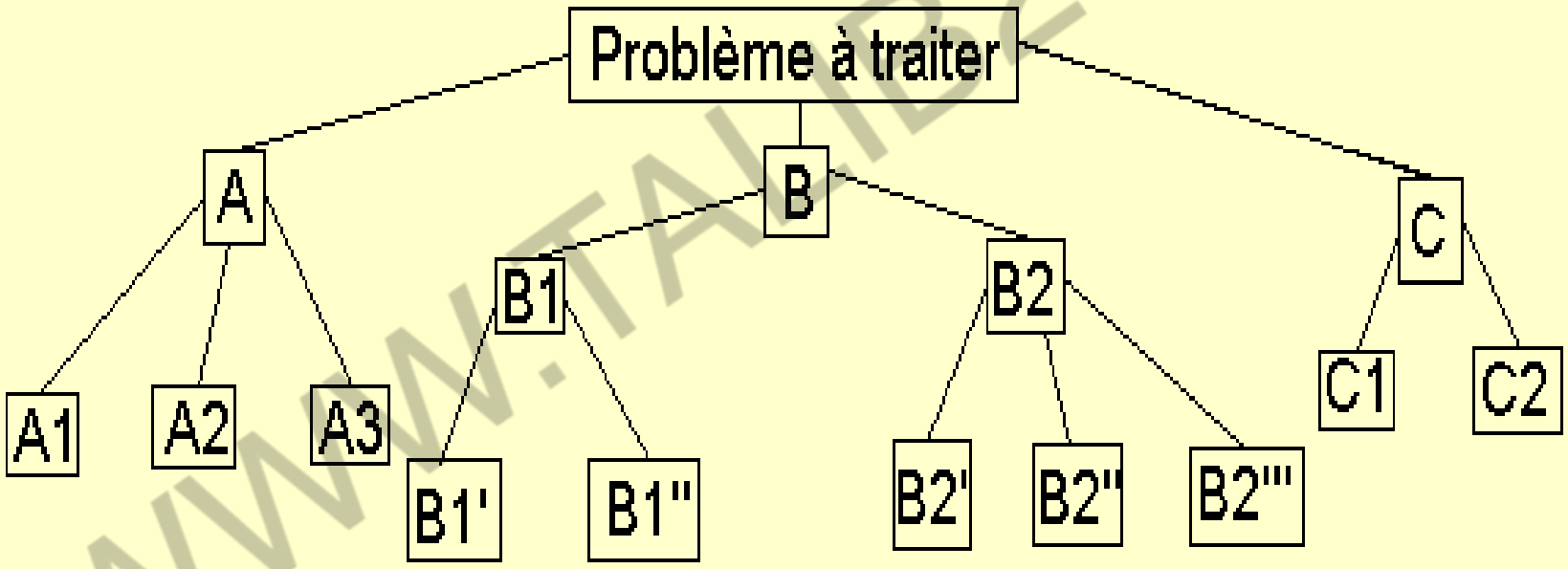
imprime à l'écran

```
i = 0
i = 1
i = 2
i = 3
valeur de i a la sortie de la boucle = 3
```

5. Les boucles imbriquées (voir TD-TP)

Démarche d'analyse descendante(Principe et exemples)

L'analyse descendante se définit comme l'approche systématique du général au particulier. Il s'agit de la classique division des problèmes.



Démarche d'analyse descendante(Principe et exemples)

Objectifs: Méthode pour écrire un algorithme de qualité.

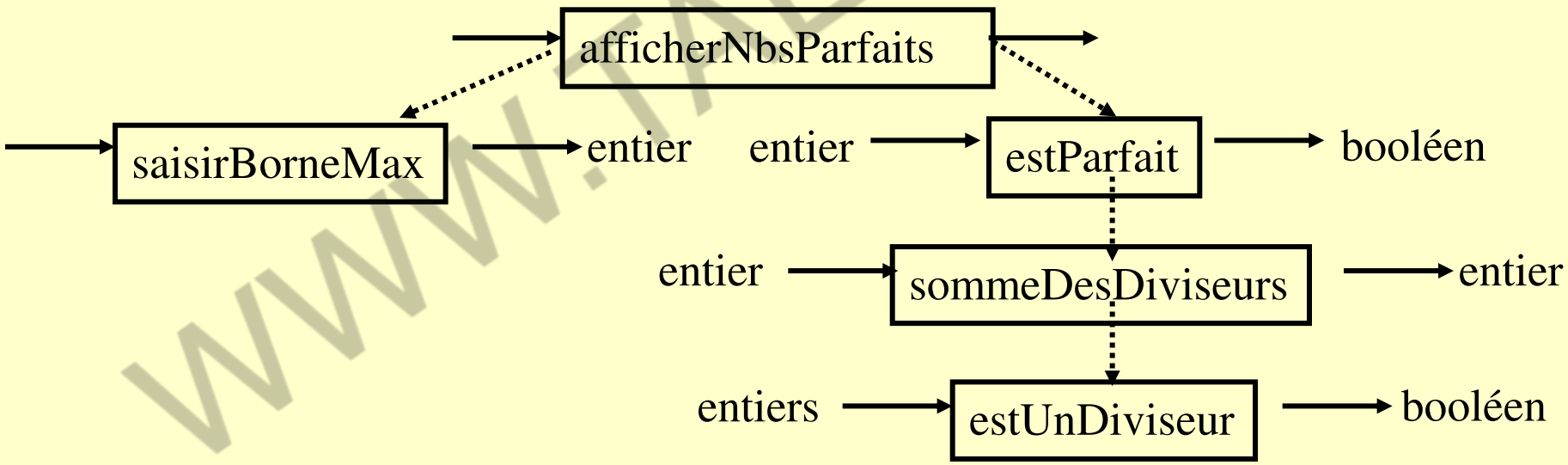
Principe:

Abstraire: Repousser le plus loin possible l'écriture de l'algorithme.

Décomposer: Décomposer la résolution en une suite de "sous-problèmes" que l'on considère comme résolus.

Combiner: Résoudre le pb par combinaison des abstractions des "sous-pbs".

Énoncé: Afficher les nombres parfaits (nombre égaux à la somme de leurs diviseurs) compris entre 1 et un nombre n (naturel ≥ 1) saisi par l'utilisateur.



Chapitre 6: Programmation modulaire

1. Introduction

La structuration de programmes en sous-programmes se fait en C à l'aide de ***fonctions***. Les fonctions en C correspondent aux fonctions *et* procédures en Pascal et en langage algorithmique. Nous avons déjà utilisé des fonctions prédéfinies dans des bibliothèques standard (**printf** de *<stdio>*, **pow** de *<math>*, etc.). Dans ce chapitre, nous allons découvrir comment nous pouvons définir et utiliser nos propres fonctions.

2. Divisions d'un programme en sous programmes:

Jusqu'ici, nous avons résolu nos problèmes à l'aide de fonctions prédéfinies et la fonction principale **main()**.

Pour des problèmes plus complexes, nous obtenons de **longues listes d'instructions**, **peu structurées** et par conséquent **peu compréhensibles**. En plus, il faut souvent répéter les **mêmes suites de commandes** dans le texte du programme!

❑ La modularité et ses avantages

Le langage C nous permet de subdiviser nos programmes en sous-programmes « fonctions » plus simples et plus compacts. A l'aide de ces structures nous pouvons ***modulariser*** nos programmes pour obtenir des solutions plus élégantes et plus efficientes.

▪ Module:

un ***module*** désigne une entité de données et d'instructions qui fournissent une solution à une (petite) partie bien définie d'un problème plus complexe. Un module peut faire appel à d'autres modules, leur transmettre des données et recevoir des données en retour. L'ensemble des modules ainsi reliés doit alors être capable de résoudre le problème global.

- **Avantages**

Voici quelques avantages d'un programme modulaire:
- Meilleure lisibilité
- Diminution du risque d'erreurs
- Possibilité de tests sélectifs
- Réutilisation de modules déjà existants
- Simplicité de l'entretien
- Favorisation du travail en équipe
- Hiérarchisation des modules

3. Procédure et fonction

La fonction est la seule sorte de module existant en C.
Dans beaucoup de langages, on trouve deux sortes de "modules", à savoir :

✓ Les **"fonctions"**, assez proches de la notion mathématique correspondante. Notamment, une fonction dispose d'arguments (en C, une fonction peut ne comporter aucun argument) qui correspondent à des informations qui lui sont transmises et elle fournit un unique résultat scalaire (simple). L'appel d'une fonction peut apparaître dans une expression.

✓ Les **"procédures"** (terme Pascal) ou "sous-programmes" (terme Fortran ou Basic) qui élargissent la notion de fonction. La procédure ne possède plus de valeur et son appel ne peut plus apparaître au sein d'une expression. Par contre, elle dispose toujours d'arguments.

3.1 Déclaration et définition de fonctions

3.1.1 Définition d'une fonction

Dans la définition d'une fonction, nous indiquons:

- le nom de la fonction
- le type, le nombre et les noms des paramètres de la fonction
- le type du résultat fourni par la fonction
- les données locales à la fonction
- les instructions à exécuter

```
<TypeRés> <NomFonct> (<TypePar1> <NomPar1>,
                      <TypePar2> <NomPar2>, ... )
{
 <déclarations locales>
 <instructions>
}
```

Remarques:

- il n'y a pas de point-virgule après la définition des paramètres de la fonction.
-Si nous choisissons un nom de fonction qui existe déjà dans une bibliothèque, notre fonction cache la fonction prédéfinie.
- Si une fonction F fournit un résultat de type T, on dit que la fonction F est de type T.

3.1.2 Déclaration d'une fonction

En C, il faut déclarer chaque fonction avant de pouvoir l'utiliser. La déclaration informe le compilateur du type des paramètres et du résultat de la fonction. A l'aide de ces données, le compilateur peut contrôler si le nombre et le type des paramètres d'une fonction sont corrects. Si dans le texte du programme la fonction est définie avant son premier appel, elle n'a pas besoin d'être déclarée.

○ Prototype d'une fonction

```
<TypeRés> <NomFonct> (<TypePar1>, <TypePar2>, ...);
```

ou bien

```
<TypeRés> <NomFonct> (<TypePar1> <NomPar1>,
                      <TypePar2> <NomPar2>, ... );
```

3.2 Arguments muets et arguments effectifs

- Les noms des arguments figurant dans l'en-tête de la fonction se nomment des "arguments muets" (arguments formels ou paramètres formels)
- les arguments fournis lors de l'appel de la Fonction se nomment des « arguments Effectifs » (ou paramètres effectifs).

3.3 L'instruction *return*

- L'instruction return renvoie un résultat sous forme de valeur de la fonction dans laquelle elle s'exécute, mais, en même temps elle interrompt l'exécution de la fonction en revenant dans la fonction qui l'a appelée.
- L'instruction return peut évaluer n'importe quelle expression.

L'instruction return peut apparaître à plusieurs reprises dans une fonction:

```
double absom (double u, double v)
{ double s ;
s = a + b ;
if (s>0) return (s) ;
else return (-s);}
```

3.4 les fonctions sans valeur de retour ou sans arguments

• Quand une fonction ne renvoie pas de résultat, on le précise, à la fois dans l'en-tête et dans sa déclaration, à l'aide du mot clé ***void***.

Par exemple, voici l'en-tête d'une fonction recevant un argument de type int et ne fournissant aucune valeur :

void sansval (int n)

Déclaration :

void sansval (int) **;**

• Quand une fonction ne reçoit aucun argument, on place le mot clé *void* à la place de la liste d'arguments. Voici l'en-tête d'une fonction ne recevant aucun argument et renvoyant une valeur de type *float.*

float tirage (**void**)

Déclaration

float tirage (**void**) ;

• Il est possible de réaliser une fonction ne possédant ni arguments ni valeur de retour. Dans ce cas, son en-tête sera de la forme:

void message (**void**)

et sa déclaration sera:

void message (**void**) ;

Remarque :

La fonction main est une fonction sans argument qui *retourne* toujours une valeur de type entier. Elle devrait donc avoir pour en-tête **int main (void)**

3.5 Variable locale et variable globale

3.5.1 Variable locale

Les variables déclarées dans un bloc d'instructions sont uniquement visibles à l'intérieur de ce bloc. On dit que ce sont des **variables locales** à ce bloc.

Ceci est vrai pour tous les blocs d'instructions, non seulement pour les blocs qui renferment une fonction. Ainsi, le bloc d'instructions d'une commande **if**, **while** ou **for** peut théoriquement contenir des déclarations locales de variables et même de fonctions.

Exemple 1

La variable NOM est définie localement à l'intérieur de la fonction HELLO. Ainsi, aucune autre fonction n'a accès à la variable NOM:

```
void HELLO(void);
{
 char NOM[20];
 printf("Introduisez votre nom : ");
 gets(NOM);
 printf("Bonjour %s !\n", NOM);
}
```

Exemple 2

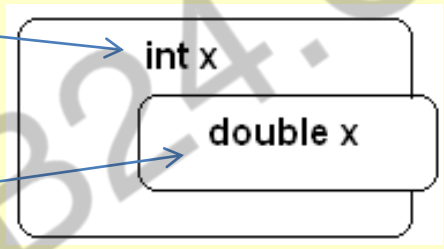
La déclaration de la variable I se trouve à l'intérieur d'un bloc d'instructions conditionnel. Elle n'est pas visible à l'extérieur de ce bloc, ni même dans la fonction qui l'entoure.

```
if (N>0)
{ int I;     for (I=0; I<N; I++)
             ... }
```

Remarque :

Une variable déclarée à l'intérieur d'un bloc ***cache*** toutes les variables du même nom des blocs qui l'entourent. Dans la fonction suivante,

```
int FONCTION(int A);
{
 int X;
 ...
 X = 100;
 ...
 while (A>10)
    {
     double X;
     ...
     X *= A;
     ...
    }
}
```



la première instruction **X=100** se rapporte à la variable du type **int** déclarée dans la fonction; l'instruction **X*=A** agit sur la variable de type **double** déclarée dans la boucle **while**. A l'intérieur de la boucle, il est impossible d'accéder à la variable locale X à la fonction.

3.5.2 Variable globale

Les variables déclarées au début du fichier, à l'extérieur de toutes les fonctions sont disponibles à toutes les fonctions du programme. Ce sont alors des **variables globales**.

En général, les variables globales sont déclarées immédiatement après les instructions **#include** au début du programme.

Exemple

La variable STATUS est déclarée globalement pour pouvoir être utilisée dans les procédures A et B.

```
#include <stdio.h>
int STATUS;

void A(...)
{
 ...
 if (STATUS>0)
      STATUS--;
 else
      ...
 ...
}

void B(...)
{
 ...
 STATUS++;
 ...
}
```

Remarque:

Les variables déclarées au début de la fonction principale *main ne sont pas* des variables globales, mais elles sont locales à *main*.

Conseils :

- Les variables globales sont à utiliser avec précaution, puisqu'elles créent des *liens invisibles entre les fonctions*. La modularité d'un programme peut en souffrir et le programmeur risque de perdre la vue d'ensemble.
- Il faut faire attention à ne pas cacher involontairement des variables globales par des variables locales du même nom.
- Il est conseillé *d'écrire des programmes aussi localement que possible*.

L'utilisation de variables globales devient inévitable, si

- Plusieurs fonctions qui ne s'appellent pas ont besoin des mêmes variables,
- Plusieurs fonctions d'un programme ont besoin du même ensemble de variables.

Ce serait alors trop gênant de passer toutes les variables comme paramètres d'une fonction à l'autre.

- Remarque: le passage par adresse sera traité dans le chapitre des pointeurs.

Prenons l'exemple de la fonction **factorielle** :

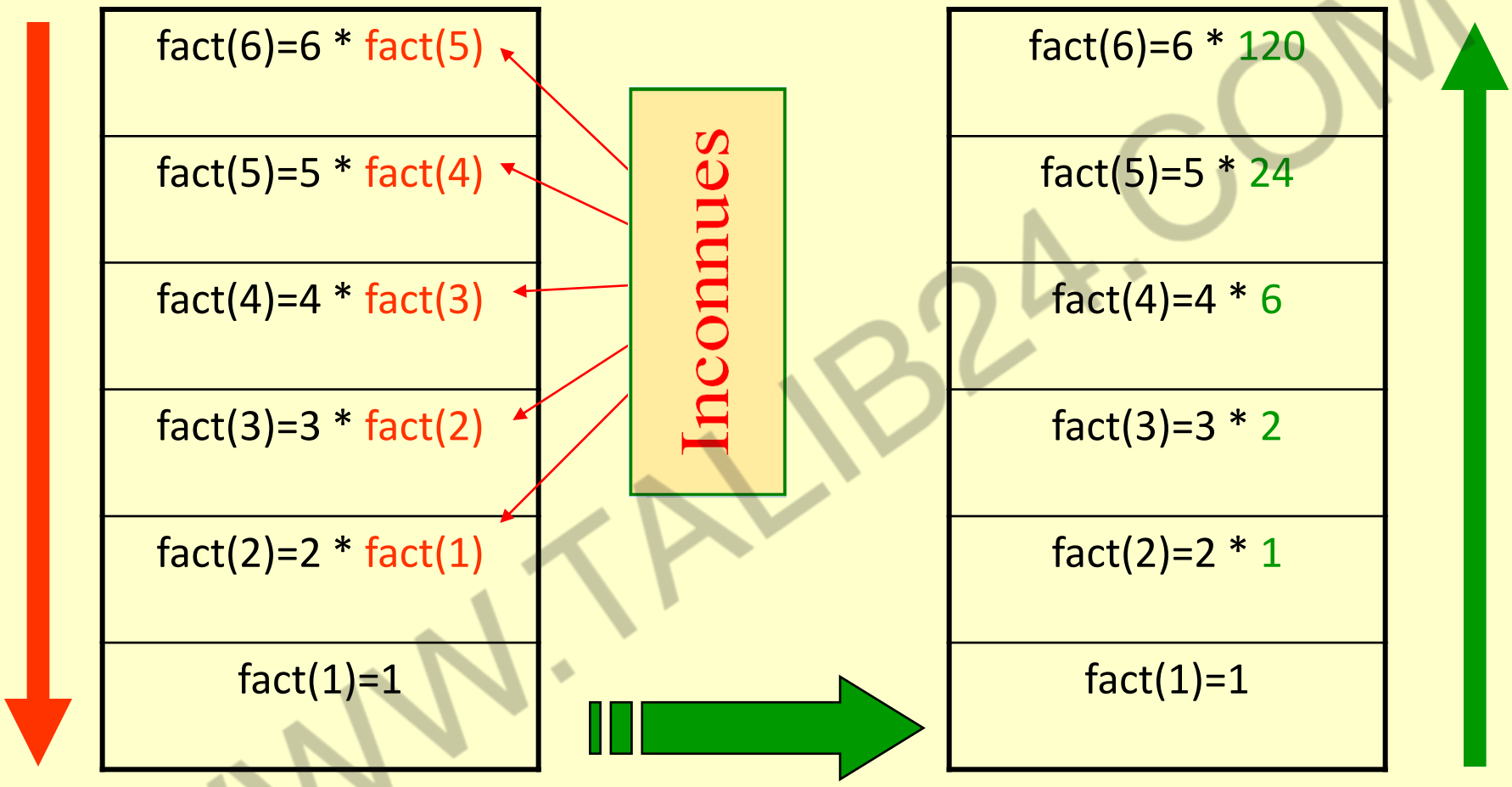
- en mathématiques :

$$n! = n.(n-1)! \quad \text{pour } n \geq 1$$

avec 0!=1

- en C on peut écrire :

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



On remarque bien les appels récursifs.
Pour calculer factorielle de i il faut attendre le calcul des factoriels des j qui lui sont strictement inférieurs. Donc on descend et on remonte !

1. Tableaux

1.1 Tableaux à une dimension

Comme tous les langages, C permet d'utiliser des "tableaux". On nomme ainsi un ensemble d'éléments de même type désignés par un identificateur unique; chaque élément est repéré par un "indice" précisant sa position au sein de l'ensemble (exemple d'un vecteur).

1.1.1 Déclarer un tableau

syntaxe : *type_elements nom_tableau[nb_cases] ;*

1.1.2 Affecter des valeurs dans des cases

syntaxe : *nom_tableau[numero_case] = valeur ;*

La numérotation des cases s'effectue de **0** à **nb_cases-1**.

1.1.3 Accéder à la valeur d'une case d'un tableau

syntaxe : *nom_tableau[numero_case]*

1.1.4 Initialisation des tableaux

Il est possible lors de la déclaration d'initialiser les cases du tableau en donnant des valeurs entre accolades.

syntaxe : type nom_tableau[N] = { val1, val2, ..., valN } ;

Exemple d'utilisation d'un tableau en C

Déterminer, à partir de vingt notes d'élèves (fournies en données), combien d'entre elles sont supérieures à la moyenne de la classe.

```
#include <stdio.h>
#define n   20
main()
{ int i, som, nbm ;
float moy ;
int t[n] ;
for (i=0 ; i<n ; i++)
{ printf ("donnez la note numéro %d : ", i+1) ;
scanf ("%d", &t[i]) ;
}
for (i=0, som=0 ; i<n ; i++) som += t[i] ;
moy = som*1.0 / n ;
printf ("\n\n moyenne de la classe : %f\n", moy) ;
for (i=0, nbm=0 ; i<n ; i++ )
if (t[i] > moy) nbm++ ;
printf ("%d élèves ont plus de cette moyenne", nbm) ;
}
```

1.2 Tableaux à deux dimensions

1.2.1 Déclarer un tableau à deux dimensions

syntaxe : *type_elements nom_tableau[taille_dim1][taille_dim2] ;*

on peut imaginer le tableau sous la forme d'un rectangle avec taille_dim1 qui représente le nombre de lignes et taille_dim2 qui représente le nombre de colonnes.

1.2.2 Affecter des valeurs dans des cases

syntaxe : nom_tableau[numero_case_dim1][numero_case_dim2] = valeur ;

Si on représente ce tableau sous la forme de lignes et de colonnes, la numérotation des cases s'effectue de 0 à numero_ligne-1 pour les lignes et de 0 à numero_colonne-1 pour les colonnes.

1.2.3 Accéder à la valeur d'une case d'un tableau à deux dimensions

syntaxe : nom_tableau[numero_case_dim1][numero_case_dim2]

1.2.4 Initialisation des tableaux à deux dimensions

syntaxe :

type nom_tableau[N][P] = {val1, val2, ..., valN+P} ;

ou

type nom_tableau[N][P] = { {valN1_1, valN1_2, ..., valN1_P}, {valN2_1, valN2_2, ..., valN2_P} ..., {valN_1, valN_2, ..., valN_P}};

2. Chaînes de caractères

Une chaîne de caractères est traitée comme un tableau à une dimension de caractères (vecteur de caractères).

La représentation interne d'une chaîne de caractères est terminée par le symbole **'\0'** (NUL). Ainsi, pour un texte de **n** caractères, nous devons prévoir **n+1** octets.

"bonjour" est équivalent au tableau

b	o	n	j	o	u	r	\0

2.1 Déclarer une chaîne de caractères

char nom_chaine[taille_chaine] ;

Exemple:

char NOM [20];

char PRENOM [20];

2.2 Affecter une chaîne lue sur l'entrée standard à une chaîne de caractères

Attention, tant que nous parlions de valeurs scalaires (uniques), nous utilisions

scanf(format, &variable);

dans le cas de chaînes de caractères, chaque caractère correspond à une case du tableau et le nom d'une chaîne de caractères est le représentant de l'adresse du premier caractère de la chaîne, **il ne doit pas être précédé de l'opérateur adresse '&'**

scanf("%s", chaine);

2.3 Initialisation de chaînes de caractères

• En général, les tableaux sont initialisés par l'indication de la liste des éléments du tableau entre accolades: **char CHAINE[] = {'H','e','l','l','o','\0'};**

• Pour les chaînes de caractères, nous pouvons utiliser une initialisation plus confortable en indiquant simplement une chaîne de caractère constante: ***char CHAINE[] = "Hello";***

• Lors de l'initialisation par **[]**, l'ordinateur réserve automatiquement le nombre d'octets nécessaires pour la chaîne, c.-à-d.: le nombre de caractères **+ 1** (ici: 6 octets).

Remarques:

- Dans les chaînes de caractères, nous pouvons utiliser toutes les séquences d'échappement définies comme caractères constants: **"Ce \ntexte \nsera réparti sur 3 lignes."**

-Le symbole **"** peut être représenté à l'intérieur d'une chaîne par la séquence d'échappement **\"**: "Affichage de \"guillemets\" \n"

-**"x" :** chaîne *de caractères* qui contient deux caractères: la lettre **'x'** et le caractère NUL: '**\0**'; **'x'** est codé dans un octet ; **"x"** est codé dans deux octets

2.4 Accès aux éléments d'une chaîne de caractères

L'accès à un élément d'une chaîne de caractères peut se faire de la même façon que l'accès à un élément d'un tableau. En déclarant une chaîne par: **char A[6];**
nous avons défini un tableau A avec six éléments, auxquels on peut accéder par: **A[0], A[1], ... , A[5]**

Exemple:

```
char A[6] = "Hello";
```

A:	'H'	'e'	'l'	'l'	'o'	'\0'
	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]

3. Lecture et affichage de chaînes de caractères

la bibliothèque *<stdio.h>* nous offre des fonctions qui effectuent l'entrée et la sortie des données. A côté des fonctions **printf** et **scanf**, nous y trouvons les deux fonctions **puts** et **gets,** spécialement conçues pour l'écriture et la lecture de chaînes de caractères.

3.1 La fonction puts

Syntaxe: **puts(<Chaîne>)**

puts(TXT) est équivalente à **printf("%s\n",TXT);**

3.2 La fonction gets

Syntaxe: **gets(<Chaîne>)** est équivalente à scanf**("%s", Chaîne)**

gets lit une *ligne* de caractères et la copie à l'adresse indiquée par <Chaîne>. Le retour à la ligne final est remplacé par le symbole de fin de chaîne **'\0'**.

Exemple de programme :

```
#include <stdio.h>
main()
{           char nom[20], prenom[20], ville[25] ;
            printf ("quelle est votre ville : ") ;
            gets (ville) ;
            printf ("donnez votre nom et votre prénom : ") ;
            scanf ("%s %s", nom, prenom) ;
            printf ("bonjour cher %s %s de ", prenom, nom) ;
            puts (ville) ;
}
```

quelle est votre ville : Casablanca
donnez votre nom et votre prénom : Zitouni Achraf
bonjour cher Zitouni Achraf de Casablanca

4. Fonctions pour le traitement de chaînes de caractères

4.1 Les fonctions de string.h

La bibliothèque ***string.h*** fournit une multitude de fonctions pratiques pour le traitement de chaînes de caractères. Voici une brève description des fonctions les plus fréquemment utilisées.

4.1.1 strlen(chaîne)

fournit la longueur de la chaîne sans compter le '\0' final

Exemple:

Strlen("bonjour") vaut 7

4.1.2 les fonctions de concaténation de chaînes

❑ **strcat(chaîne1, chaîne2)** ajoute chaîne2 à la fin de chaîne1

Exemple:

```
#include <stdio.h>
#include <string.h>
main()
{
char ch1[20] = "bonjour" ;
char ch2 [30] = " monsieur" ;
printf ("avant : %s\n", ch1) ;
strcat (ch1, ch2) ;
printf ("après : %s", ch1) ;
}
```

Espace

avant : bonjour
après : bonjour monsieur

❑ **strncat(chaîne1, chaîne2, n)**

ajoute au plus n caractères de chaîne2 à la fin de chaîne1

Exemple:

```
main()
{ char ch1[20] = "bonjour" ;
char ch2 [30] = " monsieur" ;
printf ("avant : %s\n", ch1) ;
strncat (ch1, ch2, 6) ;
printf ("après : %s", ch1) ;
}
```

avant : bonjour
après : bonjour monsi

4.1.3 les fonctions de comparaison de chaînes

❑ **strcmp(chaîne1, chaîne2)** compare chaîne1 et chaîne2 lexico graphiquement et fourni un résultat:

négatif	si chaîne1 précède chaîne2
zéro	si chaîne1 est égal à chaîne2
positif	si chaîne1 suit chaîne2

Exemple: strcmp ("bonjour", "monsieur") est négatif
strcmp ("paris2", "paris10") est positif

❑ **strncmp(chaîne1, chaîne2, n)** *(string.h)* comme strcmp mais elle limite la comparaison au nombre **n** caractères.

Exemple: strncmp ("bonjour", "bon", 4) est positif
strncmp ("bonjour", "bon", 2) vaut zéro

4.1.4 les fonctions de copie de chaînes

❑ **strcpy (cahîne1, chaîne2)**

recopie la chaîne située à l'adresse chaîne2 dans l'emplacement d'adresse chaîne1.

❑ **strncpy (chaîne1, chaîne2, n)**

procède de manière analogue à strcpy, en limitant la recopie au nombre de caractères précisés par l'expression entière **n**.

Remarque: si la longueur de la Chaîne2 est inférieure à **n**, son caractère de fin (\0) Sera recopié. Mais, dans le cas contraire, il ne le sera pas.

Tri à bulles

Le principe du tri à bulles consiste à comparer deux à deux les éléments consécutifs d'un tableau et d'effecteur une permutation s'il ne sont pas ordonnés. On continue de trier jusqu'à ce qu'il n'y ait plus de permutation à faire.

La fonction (procédure) permuter:

Entrées: - un tableau T de taille n
- deux positions i et j.

Sortie: le tableau T après permutation des cases T[i] et T[j]

Début
C←T[i]
T[i]←T[j]
T[j]←C
Fin

TRI À BULLES

<u>Algorithme Tri à bulles:</u>

Entrées: un tableau T de taille n

Sorties: le tableau T trié

Début

```
Pour i:=1 à n-1 faire:
      Pour j:=1 à n-1 faire:
            Si T[j]>T[j+1] alors:
                  permuter (T, j, j+1)
            Fin si
      Fin pour
Fin pour
```

Fin

3	7	2	6	5	1	4
3	2	6	5	1	4	7
2	3	5	1	4	6	7
2	3	1	4	5	6	7
2	1	3	4	5	6	7
1	2	3	4	5	6	7
1	2	3	4	5	6	7

TRI À BULLES

La fonction permuter en langage C:

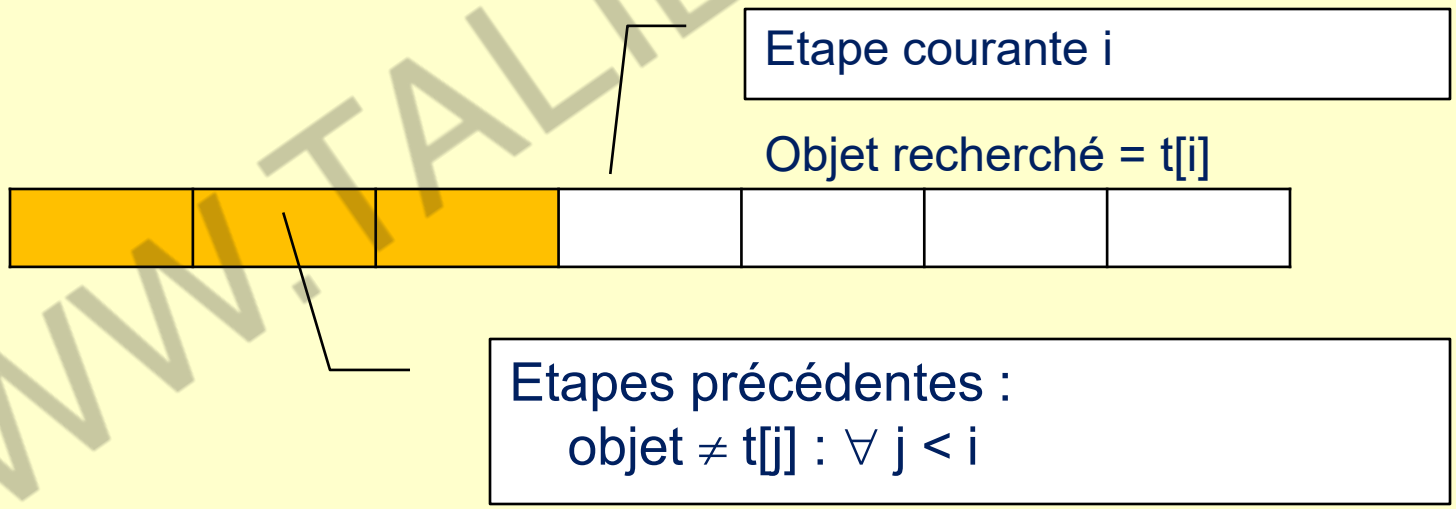
```
Void permuter(float T[], int i, int j)
    {float c=T[i];
            T[i]=T[j];
            T[j]=c;
    }
```

La fonction Tri à Bulle en langage C:

```
Void TriAbulles(float T[], int N)
    {int i,j;
     for (i=0;i<N-1;i++)
          for(j=0;j<N-1;j++)
               if(T[j]>T[j+1])
                    {permuter(T,j,j+1);}
    }
```

Recherche séquentielle (linéaire)

- Principe : comparer les uns après les autres tous les éléments du tableau avec l'objet recherché. Arrêt si :
  - l'objet a été trouvé
  - tous les éléments ont été passés en revue et l'objet n'a pas été trouvé



Recherche séquentielle (linéaire)

Fonction RechSequentielle:

Entrées: -T un tableau de taille N
- ObjRrecherche: l'élément à chercher

Sortie: La position de la première occurrence de ObjRrecherche dans T s'il existe dans T, sinon la sortie = -1

```
Début

        Pour i=1 à N faire:
                Si (t[i] == ObjRrecherche) alors:
                        Retourner(i)
                Fin Si

        Retourner (-1);

Fin
```

Recherche séquentielle (linéaire)

```
int RechSequentielle (float T[], float ObjRrecherche , int N)
{ int i;
  for (i=0 ; i<N ; i++)
      if(T[i]== ObjRrecherche) {return(i);}
  return(-1);
}
```

Recherche dichotomique

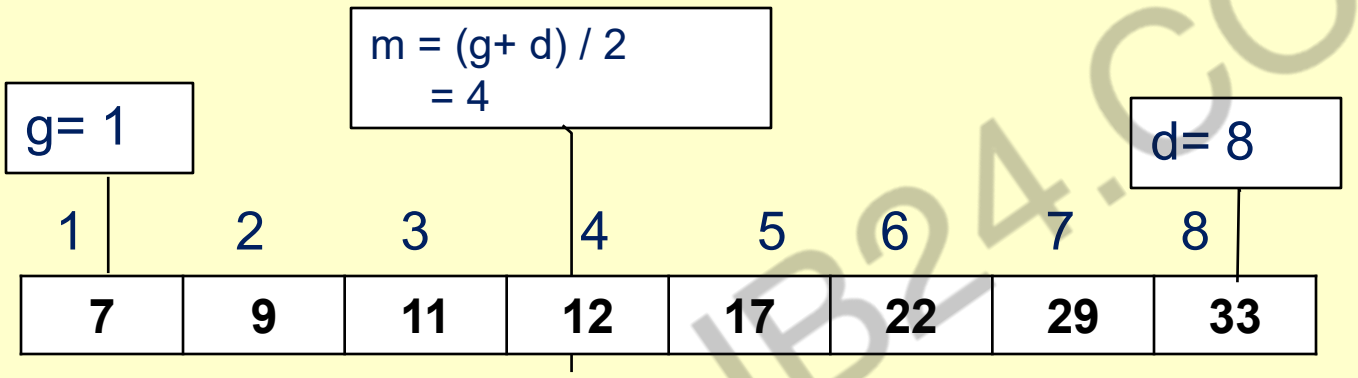
ATTENTION : il n'est applicable que sur un tableau trié.

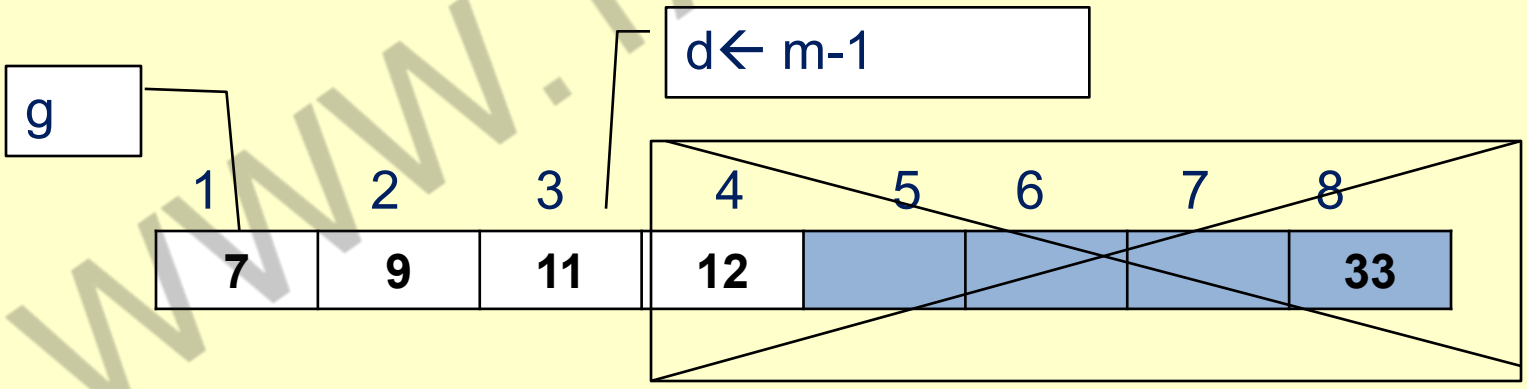
A chaque étape :

- découpage du tableau en deux sous-tableau à l'aide d'un indice milieu (tableau à gauche) et (tableau à droite)
- comparaison de la valeur située à l'indice milieu et de l'objet recherché,
  1. si l'objet est égal à la valeur T[milieu] alors il est trouvé !
  2. si l'objet est supérieur à la valeur T[milieu] relancer la recherche avec le tableau à droite,
  3. sinon relancer la recherche avec le tableau à gauche.

Exemple:

ObjRrecherche = 9





Recherche dichotomique

Fonction RechDichotomique:

Entrées: -T un tableau Trié de taille N
- ObjRrecherche: l'élément à chercher

Sortie: Une position de ObjRrecherche dans T s'il existe, sinon sortie = -1

```
Début
        g←1;   d←N
        Tant que (g<d) faire:
                m←(g+d)/2
                Si T[m] = ObjRrecherche alors Retourner(m)
                        Sinon Si T[m]< ObjRrecherche  alors g←m+1
                                Sinon d ←m+1
                            FinSi
                FinSi
        Fin tant Que
        Retourner (-1);
Fin
```

Recherche dichotomique

```
int RechDico(float T[], float ObjRrecherche, int N)
{int i,g=0,d=N-1,m;
      while(g<d)
            {m=(g+d)/2;
            if(T[m]== ObjRrecherche)
                        {return(m);}
            else if(T[m]<ObjRrecherche)
                        {g=m+1;}
                  else {d=m-1;}
            }
return(-1);
}
```

Question: Pour un tableau T trié de taille N, Quelle est la méthode de recherche la plus rapide ? Séquentielle ou dichotomique ?

Chapitre 8: Les pointeurs

1. Introduction

Toute variable manipulée dans un programme est stockée quelque part en mémoire centrale. Cette mémoire est constituée d'octets qui sont identifiés par un numéro qu'on appelle **adresse mémoire**. Pour retrouver une variable, il suffit donc de connaître l'adresse de l'octet où elle est stockée (ou, s'il s'agit d'une variable qui recouvre plusieurs octets contigus, l'adresse du premier de ces octets). Pour des raisons de lisibilité, on désigne souvent les variables par des identificateurs, et non par leur adresse. C'est le compilateur qui fait alors le lien entre l'identificateur d'une variable et son adresse en mémoire. Toutefois, il est parfois très pratique de manipuler directement une variable par son adresse.

2. Adresse mémoire et valeur d'un objet

On appelle **Lvalue** (left value) tout objet pouvant être placé à gauche d'un opérateur d'affectation. Une Lvalue est caractérisée par :

- **son adresse**: l'adresse mémoire à partir de laquelle l'objet est stocké ;
- **sa valeur**: ce qui est stocké à cette adresse.

Dans l'exemple,

```
int i, j;
i = 3;
j = i;
```

Si le compilateur a placé la variable i à l'adresse 4831836000 en mémoire, et la variable j à l'adresse 4831836004, on a

objet	adresse	valeur
i	4831836000	3
j	4831836004	3

- Deux variables différentes ont des adresses différentes. L'affectation i = j; n'opère que sur les valeurs des variables.
- Les variables i et j étant de type int, elles sont stockées sur 4 octets. Ainsi la valeur de i est stockée sur les octets d'adresse 4831836000 à 4831836003.
- L'adresse d'un objet étant un numéro d'octet en mémoire, il s'agit d'un entier (entier long) quelque soit le type de l'objet considéré. Le format interne de cet entier (16 bits, 32 bits ou 64 bits) dépend des architectures.
- L'opérateur & permet d'accéder à l'adresse d'une variable. Toutefois &i n'est pas une Lvalue mais une constante : on ne peut pas faire figurer &i à gauche d'un opérateur d'affectation.
- Pour pouvoir manipuler des adresses, on doit donc recourir un nouveau type d'objets, ce sont les pointeurs.

3. Notion de pointeur

Un pointeur est un objet (Lvalue) dont la valeur est égale à l'adresse d'un autre objet. On déclare un pointeur par l'instruction :

type *nom-du-pointeur;

Où *type* est le type d'objet pointé.

Cette « déclaration » déclare un identificateur, nom-du-pointeur, associé à un objet dont la valeur est l'adresse d'un autre objet de type **type**. L'identificateur nom-du-pointeur est un identificateur d'adresse. Comme pour n'importe quelle Lvalue, sa valeur est modifiable.

Exemple 1:

Dans l'exemple suivant, on définit un pointeur p qui pointe vers un entier i :

```
int i = 3;
int *p;
p = &i;
```

On se trouve dans la configuration

objet	adresse	valeur
i	4831836000	3
p	4831836004	4831836000

L'opérateur unaire d'indirection * permet d'accéder directement à la valeur de l'objet pointé. Ainsi, si p est un pointeur vers un entier i, *p désigne la valeur de i.

Exemple 2:

```
main()
{
int i = 3;
int *p;
p = &i;
printf("*p = %d \n",*p);
}
```

Ce programme imprime à l'écran *p=3

Dans ce programme, les objets i et *p sont identiques : ils ont mêmes adresse et valeur. Cela signifie que toute modification de *p modifie i.

Nous sommes dans la configuration :

objet	adresse	valeur
i	4831836000	3
p	4831836004	4831836000
*p	4831836000	3

Dans un programme, on peut manipuler à la fois p et *p. Ces deux manipulations sont très différentes. Comparons par exemple les deux programmes suivants:

```
main()
{
int i = 3, j = 6;
int *p1, *p2;
p1 = &i;
p2 = &j;
*p1 = *p2;
}
```

et

```
main()
{
int i = 3, j = 6;
int *p1, *p2;
p1 = &i;
p2 = &j;
p1 = p2;
}
```

Avant la dernière affectation de chacun de ces programmes, on est dans une configuration de type :

objet	adresse	valeur
i	4831836000	3
j	4831836004	6
p1	4831835984	4831836000
p2	4831835992	4831836004

Après l'affectation *p1 = *p2; du premier programme, on a

objet	adresse	valeur
i	4831836000	6
j	4831836004	6
p1	4831835984	4831836000
p2	4831835992	4831836004

Par contre, l'affectation p1 = p2 du second programme, conduit à la situation :

objet	adresse	valeur
i	4831836000	3
j	4831836004	6
p1	4831835984	4831836004
p2	4831835992	4831836004

4. Arithmétique sur des pointeurs

La valeur d'un pointeur étant un entier, on peut lui appliquer un certain nombre d'opérateurs arithmétiques classiques. Les seules opérations arithmétiques valides sur les pointeurs sont :

- l'**addition** d'un entier à un pointeur. Le résultat est un pointeur de même type que le pointeur de départ ;
- la **soustraction** d'un entier à un pointeur. Le résultat est un pointeur de même type que le pointeur de départ ;
- la **différence** de deux pointeurs pointant tous deux vers des objets de même type. Le résultat est un entier.

Remarque: la somme de deux pointeurs n'est pas autorisée.

- Si **i** est un entier et **p** est un pointeur sur un objet de type **type**, l'expression **p + i** désigne un pointeur sur un objet de type **type** dont la valeur est égale à la valeur de p incrémentée de **i * sizeof(type)**. Il en va de même pour la soustraction, et pour les opérateurs ++ et --.
- Si **p** et **q** sont deux pointeurs sur des objets de type type, l'expression p - q désigne un entier dont la valeur est égale à **(p - q)/sizeof(type)**.

Exemple: le programme suivant

```
main() {
int i = 3;
int *p1, *p2;
p1 = &i;
p2 = p1 + 1;
printf("p1 = %ld \t p2 = %ld\n",p1,p2);
}
```

Affiche p1= 483183584 p2 = 483183588

Par contre, le même programme avec des pointeurs sur des objets de type double :

```
main()
{
double i = 3;
double *p1, *p2;
p1 = &i;
p2 = p1 + 1;
printf("p1 = %ld \t p2 = %ld\n",p1,p2);
}
```

Affiche p1= 483183584 p2 = 483183592

- Les opérateurs de comparaison sont également applicables aux pointeurs, à condition de comparer des pointeurs qui pointent vers des **objets** de **même type**.
- L'utilisation des opérations arithmétiques sur les pointeurs est particulièrement utile pour **parcourir des tableaux**. le programme suivant imprime les éléments du tableau tab dans l'ordre croissant puis décroissant des indices.

```
#define N 5
int tab[5] = {1, 2, 6, 0, 7};
main()
{   int *p;
printf("\n ordre croissant:\n");
for (p = &tab[0]; p <= &tab[N-1]; p++)
printf(" %d \n",*p);
printf("\n ordre decroissant:\n");
For (p = &tab[N-1]; p >= &tab[0]; p--)
printf(" %d \n",*p);}
```