MASTER PROFESSIONEL LASERS, MATERIAUX, MILIEUX BIOLOGIQUES

COURS D'INFORMATIQUE: LANGAGE C

NOTES DE COURS

Christine ANDRAUD 2007/08

PLAN DU COURS:

I. Introduction

II. Notions de bases

III. Types de base, opérateurs, expressions

IV. Lecture et écriture de données

V. La structure alternative

VI. La structure répétitive

VII. Les tableaux

VIII. Les chaînes de caractères

IX. Les pointeurs

X. Les fonctions

XI. Les fichiers séquentiels

XII. Les "plus"

Chapitre I – Introduction

Le langage C a connu une croissance en popularité énorme ces dernières années.
On trouve ses sources en 1972, dans les laboratoires Bell, afin de développer une version portable du système d'exploitation unix. C'est un langage de programmation structuré, mais très "près" de la machine.
Publication en 1978 de "The C programming language" par Kernighan et Ritchie: définition classique du C.
Le développement de compilateurs C par d'autres maisons ont rendu nécessaire la définition d'un standard précis: le standard ANSI-C.
1983: Développement par AT&T du C++
1988: Seconde édition du livre "The C programming language"
1990: Standard ANSI-C++
Le succès du C est dû aux faits que:
- C'est un langage universel: C n'est pas orienté vers un domaine d'applications spécifique (au contraire du FORTRAN: applications scientifiques, COBOL: applications commerciales).
- C'est un langage compact: C est basé sur un noyau de fonctions et d'opérateurs limités, permettant la formulation d'expressions simples et efficaces.
- Il est près de la machine: comme il a été développé initialement pour programmer le système UNIX, il offre des opérateurs très proches de ceux du langage machine et des fonctions qui permettent un accès simple et direct aux fonctions internes de l'ordinateur (par exemple la mémoire).
- Il est rapide puisqu'il est près de la machine.
- Il est portable: en respectant le standard ANSI-C il est possible d'utiliser le même programme sur tout autre système d'exploitation en possession d'un compilateur C. C est devenu aujourd'hui le langage de programmation des micro-ordinateurs.
- Il est extensible: C ne se compose pas seulement des fonctions standard, le langage est animé par des bibliothèques de fonctions privées ou livrées par de nombreuses maisons de développement.

Désavantages:
- La possibilité d'expressions compactes entraîne le risque de se retrouver avec des programmes incompréhensibles (pour les autres, mais aussi pour nous-même), d'où la nécessité d'inclure des commentaires dans les programmes.
- C est langage proche de la machine, il est donc dangereux. Bien qu'il soit un langage de programmation structuré, il ne nous oblige pas à adopter un style de programmation (comme, par exemple le PASCAL). Le programmateur a donc beaucoup de libertés, mais aussi des responsabilités: il doit veiller à adopter un style de programmation propre, solide et compréhensible.

Chapitre II – Notions de base

1) Bibliothèques de fonctions

La pratique du C exige l'utilisation de bibliothèques de fonctions. Ces bibliothèques sont disponibles sous forme précompilées (.lib). Afin de pouvoir les utiliser, il faut inclure des fichiers en-tête (.h) dans nos programmes. Ces fichiers contiennent les prototypes des fonctions prédéfinies dans les bibliothèques et créent un lien entre les fonctions précompilées et nos programmes.
Pour inclure les fichiers en-tête:
#include <fichier.h>
Pour le compilateur que nous utiliserons, différents types de fichiers seront identifiés par leurs extensions:
.c: fichier source
.obj: fichier compilé
.exe: fichier exécutable
.lib: bibliothèque de fonctions précompilées
.h: bibliothèque en-tête

2) Composantes d'un programme en C

- La fonction main

Elle constitue le programme principal:
main()
{
déclaration des variables
instructions
}

- Les fonctions

Type_du_resultat Nom_fonction (Type_param Nom_param,…)
{
déclaration des variables locales
instructions
}

- Les identificateurs

Les noms des fonctions et des variables en C sont composés d'une suite de lettres et de chiffres:
- le premier caractère doit être une lettre
- L'ensemble des symboles utilisables est: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, …., x, y, z, A, B, C, …., X, Y, Z, _}
- Le C distingue les minuscules et les majuscules
- La longueur des identificateurs n'est pas limitée, mais le C distingue des 31 premiers caractères

Remarque:
Il est déconseillé d'utiliser le symbole "_" comme premier caractère pour un identificateur, car il est souvent employé pour définir les variables globales de l'environnement C.

Exercice 1

- Les commentaires

Un commentaire commence toujours par les deux symboles /* et se termine par les deux symboles*/. Il est interdit d'utiliser des commentaires imbriqués.
Exemple:
/* *ceci est un commentaire correct* */
/* *ceci est* /* *évidemment* */ *incorrect* */

- Les variables

Type_variable Nom_variable

3) Premier programme en C

Le classique "bonjour" (affiche bonjour à l'écran). Pour le faire, il faut:

- Inclure les bibliothèques
- Inclure le main

```
main()
{
déclaration des variables: aucune
instruction: écrire "bonjour"
}
```

La fonction prédéfinie qui permet d'écrire à l'écran est printf, elle est contenue dans le fichier en-tête stdio.h; sa syntaxe est: printf("ce que l'on veut écrire");

Voici donc notre premier programme:

```
#include <stdio.h>
main()
{
printf("bonjour\n");          /*toute instruction se termine par un point virgule*/
}
```

Remarque: les séquences d'échappement
La suite de symboles \n à la fin de la chaîne de caractère est la notation C signifiant le passage à la ligne (n comme new ligne). Il existe en C plusieurs couples de symboles qui contrôlent l'affichage ou l'impression du texte. Les séquences d'échappement sont toujours précédées par le caractère d'échappement "\".

\t	tabulation
\n	nouvelle ligne
\b	batch (curseur arrière)
\r	return (retour au début de ligne, sans saut de ligne)
\a	attention (signal acoustique)

Si l'on veut écrire le symbole " ou \: \" et \\

Chapitre III – Types de base, opérateurs, expressions

On trouvera dans un programme des variables et des constantes, il faut fixer leurs types. Pour produire de nouvelles valeurs, les variables et les constantes peuvent être combinées à l'aide d'opérateurs dans des expressions.

1) Les types de base

En mathématiques, on distingue divers ensembles de nombres (entiers naturels, entiers relatifs, réels, complexes,…). L'ordre de grandeur des nombres est illimité, ils peuvent être exprimés sans perte de précision. Un ordinateur utilise le système binaire pour sauvegarder et calculer les nombres, il existe pour un ordinateur deux grands systèmes de nombres: les entiers et les rationnels.

- Les entiers

Définition	Description	Valeur min	Valeur max	Nombre d'octets
char	caractère	-128	127	1
short	entier court	-32768	32767	2
int	entier standard	-32768	32767	2
long	Entier long	-2147483648	2147483647	4

Si l'on ajoute le préfixe **unsigned** (non signé), les domaines sont déplacés ainsi:

Définition	Valeur min	Valeur max
unsigned char	0	255
unsigned short	0	65535
unsigned int	0	65535
unsigned long	0	4294967295

Les valeurs des limites des différents types sont indiquées dans le fichier limits.h.

- Les rationnels

Définition	Précision	Mantisse*	Valeur min	Valeur max	Nb d'octets
float	simple	6	$3{,}4.10^{-38}$	$3{,}4.10^{+38}$	4
double	double	15	$1{,}7.10^{-308}$	$1{,}7.10^{+308}$	8
long double	avancée	19	$3{,}4.10^{-4932}$	$1{,}1.10^{+4932}$	10

* La mantisse est le nombre de chiffres significatifs après la virgule.

Exemple: types float, avec mantisse de 6

$$\underbrace{\left(1{,}00001.10^{8}+850\right)}-1.10^{8}$$

$$1{,}00001.10^{8}-1.10^{8}=1000$$ 

$$\underbrace{\left(1{,}00001.10^{8}-1.10^{8}\right)}+850$$

$$1000+850=1850$$ ✔

- Les variables booléennes

Il n'existe pas de type spécifique pour les variables booléennes, tous les types de variables numériques peuvent être utilisés pour exprimer des opérations logiques:

La variable logique FAUX correspond à la valeur numérique 0. La variable logique VRAI correspond toute valeur différente de 0.

- Déclaration de variables

Syntaxe:

type nom;

On peut déclarer plusieurs variables d'un même type:
Exemple:
int a, b, c;
On peut initialiser une variable lors de sa déclaration:
Exemple:
float pi = 3.14;

2) Les opérateurs standards

- L'affectation (=)

Nom_variable=expression;
Affectation avec des valeurs constantes:
Pi = 3.1416;
Lettre = 'L';
Affectation avec des valeurs variables:
a=b;
Affectation avec des expressions: voir la suite des opérateurs

- Les opérateurs arithmétiques

+	addition
-	soustraction
*	multiplication
/	division
%	modulo (reste de la division entière) (par exemple, 5%2=1)

- Les opérateurs logiques

&&	ET
||	OU
!	NON

Les résultats des opérations logiques sont de type int: la valeur 0 correspond à la valeur booléenne FAUX, la valeur 1 correspond à la valeur booléenne VRAI.

- Les opérateurs de comparaison

==	EGALITE	!=	INEGALITE
<	INFERIEUR	<=	INFERIEUR OU EGAL
>	SUPERIEUR	>=	SUPERIEUR OU EGAL

Les résultats des opérations de comparaison sont de type int: la valeur 0 correspond à la valeur booléenne FAUX, la valeur 1 correspond à la valeur booléenne VRAI
Les opérateurs logiques considèrent toute valeur différente de 0 comme VRAI, toute valeur nulle comme FAUX.
Exemple :
32&&2.3 → 1
!65,43 → 0
0||!(32>12) → 1

Les expressions sont constituées de variable ou constantes combinées entre elles par des opérateurs.
Exemple :
*aire = pi*R*R;*
moyenne = (A+B)/2
plus_grand = (x>y) (la variable plus_grand est une variable logique qui vaut 1 (resp. 0) si x est supérieur (resp. inférieur) à y)

- L'opérateur d'affectation

i = i + 2;
En C, on peut écrire:
i+ = 2;
Pour la plupart des expressions de la forme: expr1 = (expr1) Opérateur (expr2);
Il existe une formulation équivalente:expr1 Opérateur = expr2;
Cette formulation (i+ = 2) suit la logique humaine: on ajoute 2 à i.
L'opérateur d'affectation aide le compilateur à générer un code plus efficace car expr1 n'est évalué qu'une fois.

- L'opérateur d'incrémentation et de décrémentation

i = i + 1 s'écrit: i++ ou ++i
i = i - 1 s'écrit: i-- ou --i
Les opérateurs ++ et -- sont utilisés:
- pour incrémenter ou décrémenter une variable, par exemple dans une boucle (dans ce cas, pas de différence entre la notation préfixe (++i,--i) et la notation postfixe (i++,i--).
- pour incrémenter ou décrémenter une variable et en même temps affecter sa valeur à une autre variable. Dans ce cas, on choisit entre la notation préfixe et postfixe:

X = i++; passe d'abord la valeur de i à X, puis incrémente i (le ++ est après i, on l'incrémente après)
X = i--; passe d'abord la valeur de i à X, puis décrémente i
X = ++i; incrémente d'abord i puis passe la valeur de i incrémentée à X (le ++ est avant i, on l'incrémente avant)
X = --i; décrémente d'abord i puis passe la valeur de i décrémentée à X

Exemple 1:
N = 5;
X = N++; *Résultat: X = 5 et N = 6*
Exemple 2:
N = 5;
X = ++N; *Résultat: X = 6 et N = 6*

- Priorités des opérateurs

L'ordre d'évaluation des différentes parties d'une expression correspond à celle que nous connaissons en maths.
Exemple:
A = 3;
B = 4;
*X = 2 * A + 3 * B;*
6 + 12
18

Ensuite, on affecte 18 à X
Comme en maths, si l'on veut forcer l'ordinateur à commencer par un opérateur de priorité plus faible, on entoure le terme par des parenthèses.
Exemple:
A = 3;
B = 4;
*X = 2 *(A + 3) * B;*
2 * 6
12 * 4
48

Classes de priorités:

Priorité 1 (la plus élevée)	**( )**
Priorité 2	**!, ++, --**
Priorité 3	***, /, %**
Priorité 4	**+, -**
Priorité 5	**<, <=, >, >=**
Priorité 6	**==, !=**
Priorité 7	**&&**
Priorité 8	**ll**
Priorité 9 (la plus basse)	**=, +=, -=, *=, /=, %=**

Dans chaque classe, les opérateurs ont la même priorité. Si on a une suite d'opérateurs binaires de la même classe, l'évaluation se fait en passant de la gauche vers la droite.
☝: pour les opérateurs unaires (!, ++, --) et pour les opérateurs d'affectation, l'évaluation se fait de la droite vers la gauche.
Exemple 1 :
10 + 20 + 30 – 40 sera évaluée comme suit:
10 + 20) → 30
30 + 30 → 60
60 – 40 → 20
Exemple 2 : pour A = 3 et B = 4
A = B+ = 5* sera évaluée comme suit:
B+ = 5 → B = 9
A* = 9→ A = 27
Exemple 3 : pour A = 1 et B = 4
!- - A == ++!B sera évaluée comme suit:
- - A → 0 !B → 0
!0 → 1 ++0 → 1
1 == 1 → 1
Exemple 4 : X* = Y + 1 équivalent à X = X*(Y + 1)

Exercice 2

3) Conversions de type

- Calculs et affectations

Si un opérateur a des opérandes de types différents, les valeurs des opérandes sont converties automatiquement dans un type commun. Cette conversion implicite s'effectue en général des types "plus petits" vers les types "plus larges", afin de ne pas perdre en précision.

Lors d'une affectation, la donnée à droite du signe "=" est convertie dans le type à gauche du signe "=". Dans ce cas, il peut y avoir perte de précision.
Exemple :
int a =8;
float x = 10.5;
double y;
*y = a*x;*
La valeur de a est convertie en float pour pouvoir être multipliée à x. Le résultat de la multiplication est de type float, mais avant d'être affectée à y, il est converti en double, il peut donc y avoir une perte de précision.

- Appels de fonctions

Lors de l'appel d'une fonction, les paramètres sont automatiquement convertis dans les types déclarés dans la définition de la fonction.
Exemple :
int A =8, Res;
Res = pow(A, 2);
Appel de pow(): 2 et A sont convertis en doubles, le résultat est double, il est converti en int pour être affecté à Res.

- Règles de conversion lors d'une opération avec:
  - 2 entiers:
    D'abord les types char et short sont convertis en int, ensuite, l'ordinateur choisit le plus large des deux types selon l'échelle: int, unsigned int, long, unsigned long.
  - Un entier et un rationnel:
    Le type entier est converti dans le type du rationnel
  - 2 rationnels:
    L'ordinateur choisit le plus large des deux types selon l'échelle: float, double, long double.
  - Affectation:
    Lors d'une affectation, le résultat est toujours converti dans le type de la destination.

: si le type est plus faible, il peut y avoir perte de précision.
Erreur classique:
int A = 4, B = 3;
float C;
C = A / B;
A / B est entier (A et B sont int), donc A / B = 1, converti en float, C = 1!

Exercice 3

- Conversion de type forcée

Il est possible de convertir explicitement une valeur en un type quelconque:
(type) expression

Exemple:
int A = 4, B = 3;
float C;
C = (float)A / B;
Le contenu de A reste inchangé, seule la valeur utilisée dans le calcul est convertie.

Chapitre IV – Lire et écrire des données

La bibliothèque standard <stdio.h> contient un ensemble de fonctions qui assurent la communication de la machine avec le monde extérieur.

Les fonctions les plus importantes sont:

Pour la lecture:	Pour l'écriture:
printf(): écriture formatée de données	scanf(): lecture formatée de données
putchar(): écriture d'un caractère	getchar (): lecture d'un caractère

1) Ecriture formatée de données

printf(): cette fonction est utilisée pour transférer du texte, des valeurs de variables ou des résultats d'expression vers le fichier de sortie standard stdout (par défaut l'écran).

Syntaxe:

printf("format", expr_1, expr_2)

↑ ↑ ↑

Format de représentation — Expressions ou variables dont les valeurs sont à représenter

"format" est une chaîne de caractère qui peut contenir:
- du texte
- des séquences d'échappement
- des spécificateurs de format (un spécificateur pour chaque expression)

Les spécificateurs de format: ils commencent toujours par le symbole %

Symbole	Impression comme	Type
%d ou %i	Entier relatif	int
%u	Entier naturel (non signé)	int
%o	Entier exprimé en octal	int
%x	Entier exprimé en hexadécimal	int
%f	Rationnel en notation décimale	float
%e	Rationnel en notation scientifique	float
%g	Rationnel en notation décimale/scientifique	float
%lf	Rationnel en notation décimale	double
%lg	Rationnel en notation décimale/scientifique	double
%le	Rationnel en notation scientifique	double
%c	Caractère	char
%s	Chaîne de caractère	char*

Arguments de type long:

Entiers: on utilise: %ld, %li, %lu, %lo, %lx

Rationnels: pour les "long double", on utilise: %Lf, %Lg ou %Le

Remarque:
%e, %le, %Le: représentation avec 1 chiffre (non nul) avant le point décimal
%g, %lg, %Lg: choisit la représentation la plus "économique" (la plus courte) entre la notation décimale et la notation scientifique

- Largeur minimale des entiers

Il est possible d'indiquer la largeur minimale de la valeur à afficher. Dans le champ ainsi réservé, les nombres sont justifiés à droite:

	Affichage:
printf("%4d", 1);	vvv1
printf("%4d", 123);	v123
printf("%4d", 1234);	1234

- Largeur minimale et précision pour les rationnels

La précision par défaut est de 6 décimales.
printf("%f", 12.34); Affichage:12.340000

La syntaxe est: "%n.mf" où:
n est la largeur du champ
m est le nombre de décimales

	Affichage:
printf("%10.3f", 100.123);	vvv100.123
printf("%10.f", 100.123);	vvvv100.12

2) Lecture formatée de données

scanf(): fonction symétrique de printf().

Syntaxe:

scanf("format", adr_var_1, adr_var_2)

↑ ↑ ↑

Format de lecture des données

Adresses des variables auxquelles les données sont attribuées (adresse d'une variable= nom de la variable précédé de &)

- La fonction scanf reçoit ses données à partir du fichier standard stdin (le clavier)
- La chaîne de format détermine comment les données reçues doivent être interprétées
- Les données reçues correctement sont mémorisées aux adresses indiquées par adr_var_1, adr_var_2, …

Les spécificateurs de format pour scanf sont:

Symbole	Lecture d'un(e)	Type
%d ou %i	Entier relatif	int
%u	Entier naturel (non signé)	int
%o	Entier exprimé en octal	int
%b	Entier exprimé en hexadécimal	int
%f	Rationnel en notation décimale	float
%e	Rationnel en notation scientifique	float
%g	Rationnel en notation décimale/scientifique	float
%lf	Rationnel en notation décimale	double
%lg	Rationnel en notation décimale/scientifique	double
%le	Rationnel en notation scientifique	double
%c	Caractère	char
%s	Chaîne de caractère	char*

Arguments de type long:
Entiers: on utilise: %ld, %li, %lu, %lo, %lx
Rationnels: pour les "long double", on utilise: %Lf, %Lg ou %Le

scanf("%d", &nombre); On entre au clavier 33, nombre = 33

Indication de la largeur maximale: il est possible de la spécifier, mais ceci est peu recommandé, si les chiffres d'une variable passent au-delà du champ spécifié, ils seront assimilés à la prochaine variable qui sera lue!

On peut traiter plusieurs variables avec une seule instruction scanf: Lors de l'entrée des données, une suite de signes d'espacement (espace, tab, interligne) est évaluée comme un seul espace (idem si dans la chaîne de format on tape les symboles \n, \t, \r =1 seul espace).
Exemple
int jour, mois, annee;
scanf("%d %d %d",&jour, &mois, &annee);
Les entrées suivantes sont correctes et équivalentes
24␣11␣1973↵
24→11→1973↵
24↵
11↵
1973↵

Si la chaîne de format contient aussi d'autres caractères que des signes d'espacement, alors ces symboles doivent être introduits exactement dans l'ordre indiqué.
Exemple
int jour, mois, annee;
scanf("%d/%d/%d",&jour, &mois, &annee);

Entrées acceptées	Entrées rejetées
24/11/1973↵	24␣11␣1973↵
24/011/1973↵	24␣/11/␣1973↵

scanf retourne comme résultat le nombre d'arguments correctement reçus et affectés.
Exemple
int jour, mois, annee;
recu=scanf("%d %d %d",&j, &m, &a);
si l'entrée est 24␣11␣1973↵, alors recu = 3.

3) Ecriture d'un caractère

Syntaxe:
putchar (caractere)
putchar transfère le caractère "caractere" vers le fichier de sortie standard stdout (l'écran), les arguments de putchar sont des caractères (type char, i.e. des nombres entiers entre 0 et 255).

Exemples:	Affichage
putchar('x');	*x*
putchar('?');	*?*

putchar(65); *A* (65 est le code ASCII de A)
putchar('\n'); *retour à la ligne*
putchar(A); *valeur de la variable A si c'est un char*

4) Lecture d'un caractère

Syntaxe:

getchar ()

Les valeurs retournées par getchar() sont des caractères. Le type du résultat de getchar est int.

Exemple:
int C;
C=getchar();

getchar lit les données de la zone tampon stdin (clavier) et fournit les données seulement après confirmation par "Enter".

Il existe dans la bibliothèque <conio.h> une fonction getch() qui fournit immédiatement le prochain caractère entré au clavier (sans validation).

Chapitre V – Structure alternative

Les structures de contrôle définissent l'ordre dans lequel les instructions sont effectuées. Particularité des instructions de contrôle en C: les "conditions" peuvent être des expressions qui fournissent un résultat numérique.
On rappelle que: valeur 0 -> FAUX et toute valeur ≠ 0 -> VRAI

1) if-else

Structure:

```
if(condition)                 si condition est vrai (≠0)
        {                     on exécute
        bloc d'instructions 1;  bloc d'instructions 1
        }
else                          sinon
        {                     on exécute
        bloc d'instructions 2;  bloc d'instructions 2
        }
```

Remarques :
- S'il n'y a qu'une seule instruction, les accolades sont inutiles.
- "condition" peut être:
  - une variable de type numérique
  - une expression fournissant un résultat numérique
- La partie else est facultative
- On peut imbriquer plusieurs if-else

Exemple 1 :

```
if(a>b)          si a est plus grand que b
max=a;           on affecte à max la valeur de a
else             sinon
max=b;           on affecte à max la valeur de b
```

Exemple 2 :

```
if(A-B)          si A-B est vrai, si A-B est différent de 0, si A est différent de B
printf("A est différent de B\n");
else
printf("A est différent de B\n");
```

Remarque sur les exemples: Comme une seule instruction suit le if et le else, les accolades ne sont pas obligatoires.

2) if-else imbriqués

Il est possible d'imbriquer plusieurs structures if-else, cela permet de prendre des décisions entre plusieurs alternatives. Mais, afin de gagner en lisibilité, on conseille d'adopter une écriture tabulée.

Exemple :

```
if(N>0) if(A>B) MAX=A; else MAX=B;
```

Que fait ce programme? A quel if est rattaché le else?

La première remarque est qu'un else est toujours rattaché au dernier if qui ne possède pas de else, dans cet exemple, le else est donc rattaché au if (A>B). La seconde est qu'en adoptant une écriture lisible, c'est-à-dire tabulée, on verra tout de suite et beaucoup plus facilement l'imbrication des if-else.

```
if(N>0)
        if(A>B)
                MAX=A;
        else
                MAX=B;
```

Comment faire pour forcer le programme à la seconde interprétation ? (Le else est rattaché au premier if)

```
if(N>0)
        {
        if(A>B)
                MAX=A;
        }
else    MAX=B;
```

Exemple 1 :

```
int A,B ;
printf("Entrer deux nombres entiers:\n") ;
scanf("%d %d",&A,&B);
if(A>B)
    printf("%d est plus grand que %d\n",A,B);
else
    if(A<B)
        printf("%d est plus petit que %d\n",A,B);
        else
                printf("%d est égal à %d\n",A,B);
```

Affichage:

```
Entrer deux nombres entiers:
3
12
3 est plus petit que 12
```

Exemple 2 :

```
printf("Continuer (O)ui / (N)on?\n") ;
getchar(C);
if(C=='O')
    {...
    }
else
    if(C=='N')
        printf("Au revoir...\n");
        else
                printf("Erreur d'entrée\n");
```

Affichage:

```
Continuer (O)ui / (N)on?
N
Au revoir…
```

Exercices 4 et 5

3) L'opérateur conditionnel

structure:

result = expr1 ? expr2 : expr3 ;

si expr1 est vraie (≠0) result=expr2, sinon, result=expr3

Exemple :

```
MAX=(A>B)?A :B ;
```

Chapitre VI – Structure répétitive

Il existe trois sortes de structures répétitives: while, do while et for.

1) while = tant que

Structure:

```
while(condition)          tant que condition est vraie (≠0)
    {                     on exécute
    bloc d'instructions ;  bloc d'instructions
    }
```

Remarques:
- S'il n'y a qu'une seule instruction, les accolades sont inutiles.
- Le bloc d'instructions est exécuté 0 ou plusieurs fois.

Exemple 1:

```
int i =0;
while(i<4)
        {
        printf("%d\n",i);
        i++ ;
        }
```

Affichage:
0
1
2
3

Exemple 2:

```
int i =0;
while(i<4)
        printf("%d\n",i++);
```

Affichage:
0
1
2
3

Exemple 3:

```
int i =0;
while(i<4)
        printf("%d\n",++i);
```

Affichage:
1
2
3
4

Exemple 4:

```
int i =4;
while(i)
        printf("%d\n",i--);
```

Affichage:
4
3
2
1

Remarque : Le bloc d'instructions peut être vide (notation : {} ou ;), si l'on attend un évènement sans avoir besoin de traitement de données.

Exemple :
while(getch()==' '); Ignore tous les espaces entrés au clavier et sera utilisé jusqu'à l'entrée d'un caractère significatif.

2) Do-while = faire - tant que

Il est semblable au while, mais : while évalue la condition avant d'exécuter, alors que do-while exécute une fois avant d'évaluer la condition.

Structure:

```
do                              on exécute
    {
    bloc d'instructions ;       bloc d'instructions
    }
    while(condition) ;          tant que condition est vrai (≠0)
```

Remarques :

- S'il n'y a qu'une seule instruction, les accolades sont inutiles.
- Le bloc d'instructions est exécuté au moins une fois, et aussi longtemps que condition fournit une valeur vraie (différente de 0).

En pratique, do-while est moins courant que while, mais fournit parfois une solution plus élégante.

Exemple 1 :

```
int n ;
do
    {
    printf("Entrer un nombre entre 1 et 10:\n") ;
    scanf("%d",&n);
    }
    while(n<1||n>10) ;
```

```
Affichage:
Entrer un nombre entre 1 et 10:
12
Entrer un nombre entre 1 et 10:
0
Entrer un nombre entre 1 et 10:
3
```

Exemple 2 :

```
int n, div ;
printf("Entrer le nombre à diviser:\n") ;
scanf("%d",&n);
do
    {
    printf("Entrer le diviseur:\n") ;
    scanf("%d",&div);
    }
    while(!div) ;
printf("%d/%d = %f\n",n,div,(float)n/div) ;
```

```
Affichage:
Entrer le nombre à diviser:
2
Entrer le diviseur:
0
Entrer le diviseur:
3
2/3= 0.666666
```

3) for

structure:

```
for(expr1;expr2;expr3)
    {
    bloc d'instructions ;
    }
```

expr1 est évaluée une fois avant le passage dans la boucle, elle est utilisée pour initialiser les données de la boucle.

expr2 est évaluée à chaque passage de la boucle, elle est utilisée pour savoir si la boucle est répétée ou non (c'est une condition de répétition, et non d'arrêt).

expr3 est évaluée à la fin de chaque passage de la boucle, elle est utilisée pour réinitialiser les données de la boucle.

Equivalence entre une boucle for et une boucle while:

```
for(expr1;expr2;expr3)
        {
        bloc d'instructions ;
        }
```

```
expr1;
while(expr2)
        {
        bloc d'instructions ;
        expr3;
        }
```

Le plus souvent, for est utilisé comme boucle de comptage:

```
for(init;cond_repetition;compteur)
        {
        bloc d'instructions ;
        }
```

Exemple 1 :

```
int i;
for(i=1;i<=4;i++)
    {
    printf("Le carré de %d est %d\n",i,i*i);
    }
```

Affichage:
Le carré de 1 est 1
Le carré de 2 est 4
Le carré de 3 est 9
Le carré de 4 est 16

Exemple 2 :

```
int i;
for(i=4;i>0;i--)
    {
    printf("Le carré de %d est %d\n",i,i*i);
    }
```

Affichage:
Le carré de 4 est 16
Le carré de 3 est 9
Le carré de 2 est 4
Le carré de 1 est 1

Remarque :
Les parties expr1 et expr2 peuvent contenir plusieurs initialisations ou réinitialisations, séparées par des virgules.

Exemple :

```
int n,tot;
for(tot=0, n=1;n<101;n++)
        tot+=n;
printf("La somme des nombres de 1 à 100 est égale à: %d\n",tot);
```

4) Choix de la structure répétitive

On choisit la structure qui reflète le mieux l'idée du programme que l'on veut réaliser, en respectant les directives suivantes:

- Si le bloc d'instructions ne doit pas être exécuté si la condition est fausse -> while ou for
- Si le bloc d'instructions doit être exécuté au moins une fois -> do-while
- Si le nombre d'exécution du bloc d'instructions dépend de une ou plusieurs variables qui sont modifiées à la fin de chaque répétition -> for
- Si le bloc d'instructions doit être exécuté aussi longtemps qu'une condition extérieure est vraie (par exemple aussi longtemps qu'il y a des données dans un fichier d'entrées) -> while

Le choix entre for et while est souvent une question de préférence et d'habitude.

- for permet de réunir les instructions qui influencent le nombre de répétitions au début de la structure.
- while a l'avantage de correspondre plus exactement aux structures d'autres langages et à la logique humaine.
- for a le désavantage de favoriser la programmation de structures surchargées et par la suite illisibles.
- while a le désavantage de mener parfois à de longues structures dans lesquelles il faut chercher pour trouver les instructions influençant la condition de répétition.

Chapitre VII – Les tableaux

Les tableaux sont des variables structurées. Dans une première approche, leur traitement en C ne diffère pas de celui dans d'autres langages. On verra plus loin (Ch. IX – Les pointeurs) que le C permet un accès encore plus direct et rapide aux données d'un tableau.

1) Tableaux à 1 dimension

Un tableau unidimensionnel est une variable structurée formée d'un nombre entier N de variables simples du même type (composantes du tableau). Lors de la déclaration de variable, N doit être défini.

a) Déclaration et mémorisation

- Déclaration :

type Nom_du_tableau[dimension] ;

Remarque: Nom_du_tableau est un identificateur

Exemples: *int A[25];* tableau de 25 entiers de type int
float B[10]; tableau de 10 décimaux de type float
char C[30]; tableau de 30 caractères (entiers de type char)

- Mémorisation :

En C, le nom d'un tableau est le représentant de l'adresse du premier élément du tableau. Les adresses des autres composantes sont calculées automatiquement, relativement à cette adresse.

Si un tableau possède N composantes et si le type déclaré des composantes requiert M octets, la mémoire réservée pour ce tableau est de N×M octets.

Exemple:
int A[6];
Un entier int requiert 2 octets, il y a 6 éléments, la mémoire réservée pour le tableau A est donc de 2×6=12 octets.

b) Initialisation et réservation automatique

- Il est possible d'initialiser un tableau lors de sa déclaration, en indiquant la liste des valeurs
Exemple:
int A[5]={100,200,300,400,500};
Le premier élément du tableau, A[0] contiendra la valeur 100. le second élément, A[1], contiendra la valeur 200, le dernier élément, A[4], contiendra la valeur 500.

- Lors de sa déclaration, on peut initialiser le tableau, si le nombre de valeur dans la liste est inférieur à la dimension du tableau, les composantes restantes sont mises à 0.
Exemple:
int A[4]={10,20};

Le premier élément du tableau, A[0] contiendra la valeur 10. le second élément, A[1], contiendra la valeur 20, le troisième, A[2] sera mis par défaut à 0, de même pour le dernier élément, A[3].

- Si la dimension n'est pas indiquée, l'ordinateur réserve automatiquement le nombre d'octets nécessaires.

Exemple:
int A[]={10,20,30};
Les trois éléments du tableau sont initialisés à 10, 20 et 30, la taille du tableau sera automatiquement mise à 3.

Remarque: Il faut bien évidemment que le nombre de valeurs dans la liste soit inférieur ou égal à la dimension du tableau !

c) Accès aux composantes

Lorsque l'on déclare un tableau, (par exemple int A[5] ;), on définit un tableau avec 5 composantes, auxquelles on peut accéder par :
A[0], A[1], A[2], A[3], A[4]

Remarque: Le premier élément du tableau est l'élément 0, donc, pour un tableau de dimension N, le premier sera l'élément 0, le dernier l'élément N-1.

d) Affichage et affectation

La structure for se prête particulièrement bien au travail avec les tableaux.

- Affichage du contenu d'un tableau

Exemple:

```
int A[5]={1,2,3,4,5} ;
int i ;
for(i=0 ;i<5 ;i++)
        printf("%3d",A[i]);
```

Affichage:

```
 1  2  3  4  5
```

Remarque : Avant de pouvoir afficher les composantes d'un tableau, il faut bien sûr leur affecter des valeurs!

- Affectation

Exemple:

```
int A[5]={1,2,3,4,5} ;
int i ;
for(i=0 ;i<5 ;i++)
        scanf("%d",&A[i]);
```

Remarque: De la même manière que pour une variable "normale", on indique l'adresse (&A[i])

Exercice 6

2) Tableaux à 2 dimensions

En C, un tableau A à deux dimensions est à interpréter comme un tableau (à une dimension) de dimension L dont chaque composante est un tableau (unidimensionnel) de dimension C.
On appelle L le nombre de lignes, C le nombre de colonnes.
L et C sont les dimensions du tableau.
Un tableau à deux dimensions contient donc L×C composantes.
Rapprochement avec les maths: "A est un vecteur de L vecteurs de dimension C", ou encore "A est une matrice de dimensions L et C".

a) Déclaration, mémorisation, réservation automatique

- Déclaration :

type Nom_du_tableau[ligne][colonne] ;

Exemple : *int A[10][10];* tableau de 10×10 entiers de type int
float B[5][4]; tableau de 5×4 décimaux de type float
char C[2][25]; tableau de 2×25 caractères

- Mémorisation :

Comme pour les tableaux à une dimension, le nom d'un tableau est le représentant de l'adresse du 1er élément du tableau, c'est-à-dire l'adresse de la première ligne. Les composantes d'un tableau sont stockées ligne par ligne.

Exemple : *int A[3][2]={{1,2},{10,20},{100,200}};* tableau de 3×2 entiers
Si le nombre de lignes ou de colonnes n'est pas déclaré explicitement, l'ordinateur réserve automatiquement le nombre d'octets nécessaires.

Exercice 7

b) Accès aux composantes

Comme pour les tableaux à une dimension, les indices des tableaux varient de 0 à N-1.
Soit un tableau à deux dimensions A[n][m]:

Premier élément→	A[0][0]	……	A[0][m-1]	
	:		:	
	:		:	
	A[n-1][0]	……	A[n-1][m-1]	←dernier élément

c) Affichage et affectation

Lors du travail avec les tableaux à deux dimensions, on utilisera (souvent imbriqués) deux indices (par exemple i et j), et la structure for, pour parcourir les lignes et les colonnes.

- Affichage

Exemple 1 :

int A[3][2]={{1,2},{10,20},{100,200}}; *int i=2, j=1;* *printf("élément [%d][%d]= %d",i,j,A[i][j]);*	Affichage: élément[2][1]=200

Exemple 2 :

```
int A[3][2]={{1,2},{10,20},{100,200}};
int i, j;
for(i=0;i<3;i++)  /* boucle sur les lignes*/
        {
        for(j=0;j<2;j++) /* boucle sur les */
                {                /* colonnes */
                printf("%5d",A[i][j]);
                }
        printf("\n");
        }
```

Affichage:

```
    1    2
   10   20
  100  200
```

- Affectation

Exemple :

```
int A[5][4];
int i, j;
for(i=0;i<5;i++)  /* boucle sur les lignes*/
        for(j=0;j<4;j++) /* boucle sur les colonnes */
                scanf("%d",&A[i][j]);
```

Exercice 8

Exercice 9

Chapitre VIII – Les chaînes de caractères

Il n'existe pas de type particulier "chaîne" ou "string" en langage C. Une chaîne de caractères est traitée comme un tableau de caractères à une dimension.
Il existe tout de même des notations particulières et une bonne quantité de fonctions spéciales pour le traitement des tableaux de caractères.

1) Déclaration, mémorisation

a) Déclaration

char Nom_du_tableau[longueur] ;
Lors de la déclaration (comme pour les tableaux de chiffres), on doit indiquer l'espace à réserver.
La représentation interne d'une chaîne de caractères est terminée par le symbole '\0' (NUL), donc pour un texte de n caractères, il faudra réserver n+1 emplacements.
☝: Il n'y a pas de contrôle de cela à la compilation, s'il y a un problème, celui-ci ne se verra qu'à l'exécution.

b) Mémorisation

De même que pour les tableaux de chiffres, le nom de la chaîne est le représentant de son adresse mémoire, et plus précisément le représentant de l'adresse du premier caractère.
Exemple :
char *TXT[9]="BONJOUR!";*

'B'	*'O'*	*'N'*	*'J'*	*'O'*	*'U'*	*'R'*	*'!'*	*\0*

↑
TXT
TXT[0]

2) Les chaînes de caractères constantes

- Les chaînes de caractères sont indiquées entre guillemets. La chaîne de caractères vide est alors:""
- Dans les chaînes de caractères, on peut utiliser les séquences d'échappement, définies comme caractères constants

Exemple 1 :
char C[]="Ce texte\n s'écrira sur \n 3 lignes";
Exemple 2 :
char txt[]={'L','\'' ','a','s','t','u','c','e','\0'};
- Plusieurs chaînes de caractères constantes qui sont séparées par des signes d'espacement dans le texte du programme seront réunies en une seule chaîne lors de la compilation:

Exemple :
char A[]="un ""deux ""trois";
La chaîne contenue dans A sera évaluée "un deux trois". Il est ainsi possible de définir de très longues chaînes de caractères constantes utilisant plusieurs lignes dans le texte du programme.

☝ 'x' est un caractère constant (qui a une valeur numérique 'x'=120) (codé sur 1 octet)
"x" est une chaîne de caractères qui contient 2 caractères 'x' et '\0' (codé sur 2 octets)

3) Initialisation de chaînes de caractères

En général, on initialise les tableaux en indiquant la liste de ses éléments entre accolades:
char chaine[]={'h','e','l','l','o','\0'};

Mais, pour une chaîne de caractères, on peut utiliser:
char chaine[]= "hello";

Remarque : Si on ne spécifie pas la longueur, le bon nombre d'octets est réservé (y compris le '\0').

Représentation en mémoire :

char txt[]="hello";

'h'	'e'	'l'	'l'	'o'	\0

char txt[6]="hello";

'h'	'e'	'l'	'l'	'o'	\0

char txt[8]="hello";

'h'	'e'	'l'	'l'	'o'	\0	0	0

char txt[5]="hello";

'h'	'e'	'l'	'l'	'o'

Passe à la compilation, erreur à l'exécution

char txt[4]="hello";

'h'	'e'	'l'	'l'

Erreur à la compilation

Exercice 10

4) Accès aux éléments d'une chaîne

L'accès se fait de la même manière que pour un élément d'un tableau.
En déclarant:
char A[6]="hello";
on a:

'h'	*'e'*	*'l'*	*'l'*	*'o'*	*\0*

↑ ↑ ↑
A[0] A[1] A[5]

5) Travailler avec les chaînes de caractères

a) Les fonctions de stdio.h

- Affichage
  - printf (avec le spécificateur %s)

char NOM[]="Hello world";	Affichage:
printf":%s:",NOM)	:Hello world:
printf":%15s:",NOM)	: Hello world: (aligne à droite)
printf":%-15s:",NOM)	:Hello world : (aligne à gauche)
printf":%5s:",NOM)	:Hello:

- puts

La fonction puts est idéale pour écrire une chaîne de caractère ou le contenu d'une variable dans une ligne isolée.
puts(ch): écrit la chaîne de caractère ch et provoque un retour à la ligne.

Equivalence: puts(ch) = printf("%s\n",ch)

Exemple :

char texte[]="Voici une première ligne"; *puts(texte);* *puts("voici une deuxième ligne");*	Affichage: Voici une première ligne voici une deuxième ligne

- Lecture
  - scanf (avec le spécificateur %s)

Le nom de la chaîne est le représentant de l'adresse du premier caractère, il est donc inutile (et interdit!) de mettre &.

Exemple :
```
char lieu[30];
int jour, mois, annee;
printf("Entrez vos lieu et date de naissance:\n");
scanf(%s %d %d %d",lieu,&jour,&mois,&annee);
```

  - gets

gets(ch): lit une (ou plusieurs) lignes de caractères et la (les) copie à l'adresse indiquée par ch. Le retour à la ligne final est remplacé par \0
Exemple :
```
int MAX=1000;
char ligne[MAX];
gets(ligne);
```

b) Les fonctions de string.h

La bibliothèque <string.h> fournit une multitude de fonctions pratiques pour le traitement de chaînes de caractères.
Dans le tableau suivant, n représente un nombre du type int. Les symboles s et t peuvent être remplacés par:

- une chaîne de caractères constante
- le nom d'une variable déclarée comme tableau de char
- un pointeur sur char (Cf. Chapitre IX)

strlen(s)	fournit la longueur de la chaîne sans compter le '\0' final
strcpy(s, t)	copie t vers s
strcat(s, t)	ajoute t à la fin de s
strcmp(s, t)	compare s et t lexicographiquement et fournit un résultat: négatif: si s précède t zéro: si s est égal à t positif: si s suit t
strncpy(s, t, n)	copie au plus n caractères de t vers s
strncat(s, t, n)	ajoute au plus n caractères de t à la fin de s

c) Les fonctions de stdlib

La bibliothèque <stdlib.h> contient des déclarations de fonctions pour la conversion de nombres en chaînes de caractères et vice-versa.

- Chaîne -> nombre

Dans le tableau suivant, le symbole s peut être remplacé par:
- une chaîne de caractères constante
- le nom d'une variable déclarée comme tableau de char
- un pointeur sur char (Cf. Chapitre IX)

atoi(s)	retourne la valeur numérique représentée par s comme int
atol(s)	retourne la valeur numérique représentée par s comme long
atof(s)	retourne la valeur numérique représentée par s comme double (!)

Règles générales pour la conversion:
- Les espaces au début d'une chaîne sont ignorés
- Il n'y a pas de contrôle du domaine de la cible
- La conversion s'arrête au premier caractère non-convertible
- Pour une chaîne non-convertible, les fonctions retournent zéro

- Nombre -> chaîne

Le standard ANSI-C ne contient pas de fonctions pour convertir des nombres en chaînes de caractères. Si on se limite aux systèmes fonctionnant sous DOS, on peut quand même utiliser les fonctions itoa, ltoa et ultoa qui convertissent des entiers en chaînes de caractères.
Chacune de ces trois procédures convertit son premier argument en une chaîne de caractères qui sera ensuite attribuée à s. La conversion se fait dans la base b.
Dans le tableau suivant, n_int est un nombre du type int, n_long est un nombre du type long, n_uns_long est un nombre du type unsigned long, s est une chaîne de caractères (longueur maximale de la chaîne: 17 resp. 33 octets), b est la base pour la conversion (2 ... 36).

itoa (n_int, s, b)
ltoa (n_long, s, b)
ultoa (n_uns_long, s, b)

d) Les fonctions de ctype

Les fonctions de ctype servent à classifier et à convertir des caractères. Les symboles nationaux (é, è, ä, ü, ß, ç, ...) ne sont pas considérés. Les fonctions de ctype sont indépendantes du code de caractères de la machine et favorisent la portabilité des programmes. Dans la suite, c représente une valeur du type int qui peut être représentée comme caractère.
Les fonctions de classification suivantes fournissent un résultat du type int différent de zéro, si la condition respective est remplie, sinon zéro.

La fonction:	retourne une valeur différente de zéro,
isupper(c)	si c est une majuscule ('A'...'Z')
islower(c)	si c est une minuscule ('a'...'z')
isdigit(c)	si c est un chiffre décimal ('0'...'9')
isalpha(c)	si islower(c) ou isupper(c)
isalnum(c)	si isalpha(c) ou isdigit(c)
isxdigit(c)	si c est un chiffre hexadécimal ('0'...'9' ou 'A'...'F' ou 'a'...'f')
isspace(c)	si c est un signe d'espacement(' ', '\t', '\n', '\r', '\f')

Les fonctions de conversion suivantes fournissent une valeur du type int qui peut être représentée comme caractère; la valeur originale de c reste inchangée:

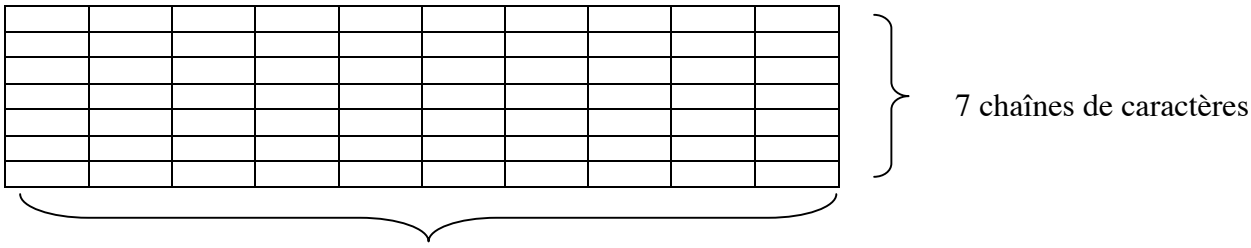
tolower(c)	Retourne c converti en minuscule si c est une majuscule
toupper(c)	Retourne c converti en majuscule si c est une minuscule

6) Tableaux de chaînes de caractères

a) Déclaration, initialisation, mémorisation

type nom[L][C];

Exemple :
char jour[7][9]; réserve l'espace en mémoire pour 7 chaînes de caractères contenant 9 caractères(8 significatifs)



Exemple :
char jour[7][9]={"lundi","mardi","mercredi","jeudi","vendredi","samedi","dimanche"};

l	u	n	d	i	\0	0	0	0
m	a	r	d	i	\0	0	0	0
m	e	r	c	r	e	d	i	\0
j	e	u	d	i	\0	0	0	0
v	e	n	d	r	e	d	i	\0
s	a	m	e	d	i	\0	0	0
d	i	m	a	n	c	h	e	\0

b) Accès aux différentes composantes

- Il est possible d'accéder aux différentes chaînes de caractères d'un tableau en indiquant la ligne correspondante.

Exemple :
char jour[7][9]={"lundi","mardi","mercredi","jeudi","vendredi","samedi","dimanche"};
printf("Aujourd'hui, c'est %s!\n",jour[2]);
affichage: Aujourd'hui, c'est mercredi!

jour[j] représente l'adresse du premier élément d'une chaîne de caractère, on ne peut donc modifier une telle adresse par une affectation du type:

~~*jour[4]="friday";*~~

On doit utiliser la fonction strcpy(jour[4],"friday")

- Il est possible d'accéder aux différents caractères qui composent le tableau:

for(i=0;i<7;i++) *printf("%c\t", jour[i][0];*	Affichage: l m m j v s d

Chapitre IX – Les pointeurs

Les pointeurs existent dans la plupart des langages de programmation, ils permettent d'accéder à la mémoire de l'ordinateur. Ce sont des variables auxquelles on peut attribuer les adresses d'autres variables.
En C, ils jouent un rôle primordial dans la définition de fonctions: comme le passage des paramètres en C se fait toujours par la valeur, les pointeurs sont le seul moyen de changer le contenu de variables déclarées dans d'autres fonctions. Ainsi, le traitement de tableaux ou de chaînes de caractères serait impossible sans l'utilisation des pointeurs.

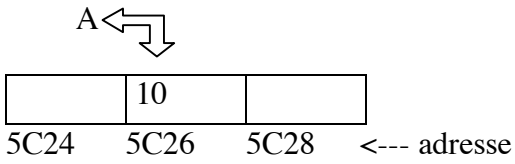
1) Adressage de variables

Il existe deux modes d'adressage principaux: l'adressage direct, et l'adressage indirect.

a) Adressage direct = Accès au contenu d'une variable par le nom de cette variable

Exemple: *int A;* On déclare une variable A,
A=10; On lui affecte la valeur 10 (son contenu vaut 10)

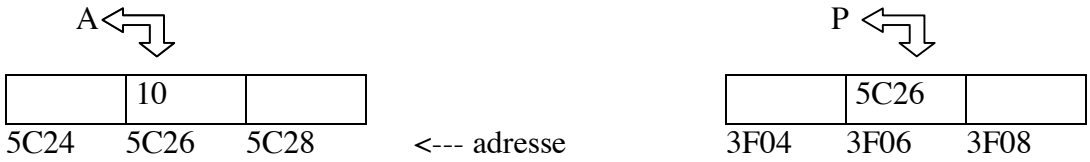
Représentation en mémoire:



b) Adressage indirect

Si on ne veut, ou on ne peut utiliser le nom d'une variable A, on peut copier l'adresse de cette variable dans une variable spéciale P, appelée pointeur. On peut ainsi, par la suite, retrouver l'information de A en passant par P.

Représentation en mémoire:
A : variable contenant la valeur 10
P : pointeur (variable) contenant l'adresse de A (&A ou 5C26)



2) Les pointeurs

Définition: Un pointeur est une variable spéciale qui peut contenir l'adresse d'une autre variable.
En C, chaque pointeur est limité à un type de données. Il peut contenir l'adresse d'une variable simple de ce type, ou l'adresse d'une composante d'un tableau de ce type.
Si un pointeur P contient l'adresse d'une variable A, on dit que P pointe sur A.

Remarques:
Pointeurs et noms de variables ont le même rôle: ils donnent accès à un emplacement dans la mémoire interne de l'ordinateur, il faut tout de même bien faire la différence:

- Un pointeur est une variable qui peut "pointer" sur différentes adresses.
- Le nom d'une variable reste toujours liée à la même adresse.

a) Les opérateurs de base

- L'opérateur "adresse de": & (pour obtenir l'adresse d'une variable)

&var fournit l'adresse de la variable var (déjà vu dans scanf)

Exemple: *P=&A;* On affecte à P l'adresse de A

- L'opérateur "contenu de": *

*pointeur: désigne le contenu de l'adresse référencée par pointeur

Exemple:
A: variable contenant la valeur 10
B: variable contenant la valeur 50
P pointeur non initialisé

P=&A;	P pointe sur A
B=*P*;*	affecte à B le contenu de l'adresse référencée par P (i.e. le contenu de A) ⇒ B=10
**P=20;*	le contenu de l'adresse référencée par P (i.e. le contenu de A) est mis à 20 ⇒ A=20

- Déclaration d'un pointeur

Syntaxe:

type *nom_pointeur	Déclare un pointeur (nom_pointeur) qui peut recevoir des adresses de variables de type "type".

Exemple: *int *p;* Déclare un pointeur p sur int

Lors de la déclaration d'un pointeur en C, ce pointeur est lié explicitement à un type de données. Ainsi, la variable P déclarée comme pointeur sur int, ne peut pas recevoir l'adresse d'une variable d'un autre type que int.

b) Les opérations élémentaires sur pointeurs

- Priorités de * (contenu de) et & (adresse de)

* et & ont la même priorité que les opérateurs unaires (!, ++, --), dans une même expression, les opérateurs unaires *, &, !, ++, -- sont évalués de droite à gauche.

- Si p pointe sur x, alors *p peut être utilisé partout où l'on peut écrire x.

Exemple:
p=&x

*y=*p+1*	⇔	y=x+1
**p=*p+10*	⇔	x=x+10
**p+=2*	⇔	x+=2
*++*p*	⇔	++x
*(*p)++*	⇔	x++

Dans ce dernier exemple, les parenthèses sont nécessaires: comme les opérateurs * (contenu de) et ++ sont évalués de droite à gauche, sans parenthèses le pointeur p serait incrémenté, et non pas l'objet sur lequel pointe p.

- Le pointeur nul: la valeur numérique 0 est utilisée pour indiquer qu'un pointeur ne pointe "nulle part".

Exemple:

```
int *p;
p=0;
```

- Les pointeurs sont aussi des variables, et peuvent être utilisés comme telles. Soient p1 et p2, pointeurs sur int, p1 = p2 copie le contenu de p2 dans p1, p1 pointe alors sur le même objet que p2.

En résumé:

int A;
int *p;
p=&A;
A désigne le contenu de A
&A désigne l'adresse de A
p désigne l'adresse de A
***p désigne le contenu A**
&p désigne l'adresse du pointeur p

Exercice 11

3) Pointeurs et tableaux

Il existe une relation très étroite entre tableaux et pointeurs, ainsi, chaque opération avec des indices de tableaux peut être exprimée à l'aide de pointeurs. En général, les versions formulées avec pointeur sont plus compactes et plus efficaces, surtout dans les fonctions (Cf. chapitre X)

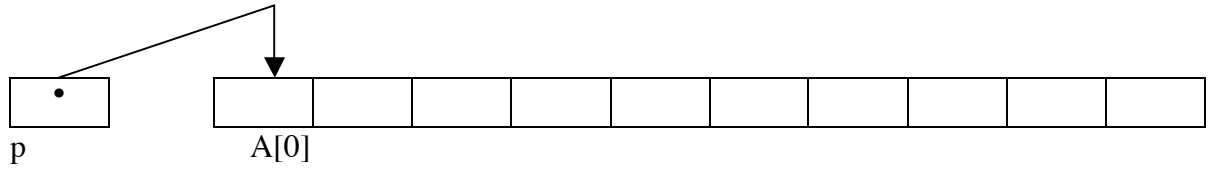
a) Adressage des composantes d'un tableau

On a vu au chapitre VII que le nom d'un tableau représentait l'adresse de son premier élément.

tableau ⇔ &tableau[0]

Exemple:

```
int A[10];
int *p;
p=A;   est équivalente à   p=&A[0];
```



Si p pointe sur une composante quelconque d'un tableau, p+1 pointe sur la composante suivante.

Exemple:
int A[10];
*int *p;*
p=A;
**(p+1)* désigne le contenu de A[1]
**(p+i)* désigne le contenu de A[i]

Il pourrait être surprenant que (p+i) n'adresse pas le i-ème octet derrière p, mais le pointeur est limité à un type de données, le compilateur connaît le nombre d'octet des différents types.

Comme A représente l'adresse de A[0]:
**(A+1)* désigne le contenu de A[1]
**(A+i)* désigne le contenu de A[i]

MAIS il existe toujours une différence essentielle entre un pointeur et le nom d'un tableau:

- Un pointeur est une variable:
  p=A ou p++ est permis
- Le nom d'un tableau est une constante:
  ~~A=p~~ ou ~~A++~~ est impossible

En résumé:
A: tableau de type quelconque (i: indice)
p: pointeur de même type que A
A désigne l'adresse de A[0]
A+i désigne l'adresse de A[i]
***(A+i) désigne le contenu A[i]**
Si p=A
p pointe sur A[0]
p+i pointe sur A[i]
***(p+i) désigne le contenu de A[i]**

On peut donc passer du formalisme "tableau" au formalisme "pointeur", en remplaçant tab[i] par *(tab+i).

b) Arithmétique des pointeurs

Le C soutient une série d'opérateurs arithmétiques sur les pointeurs que l'on ne rencontre en général que dans les langages machine.
Toutes les opérations avec les pointeurs tiennent compte du type et de la grandeur des objets pointés.

- Affectation par un pointeur de même type

p1=p2; p2 pointe sur le même objet que p1

- Addition et soustraction d'un nombre entier

Si p pointe sur A[i] (élément i d'un tableau A), alors:
p+n pointe sur A[i+n]
p-n pointe sur A[i-n]

- Incrémentation et décrémentation

Si p pointe sur A[i], alors:

p++;	p pointe sur A[i+1]
p+=n;	p pointe sur A[i+n]
p--;	p pointe sur A[i-1]
p-=n;	p pointe sur A[i-n]

MAIS: Attention à ne pas sortir du tableau!!!

- Soustraction de deux pointeurs

p1 et p2 pointent dans le même tableau
p1 - p2 < 0 si p1 précède p2
p1 - p2 = 0 si p1 = p2 (p1 pointe au même endroit que p2)
p1 - p2 > 0 si p2 précède p1
p1 - p2 est indéfini si p1 et p2 ne pointent pas dans le même tableau

De manière générale, la soustraction de deux pointeurs pointant dans le même tableau est équivalente à la soustraction des indices correspondants.

- Comparaison de deux pointeurs (<, >, <=, >=, = =, !=)

La comparaison est équivalente à la comparaison des indices, si les deux pointeurs ne pointent pas dans le même tableau, le résultat est donné par leurs positions relatives dans la mémoire de l'ordinateur.

Exercice 12

c) Pointeurs et chaînes de caractères

De la même manière que pour les int (ou float, ou double, …), un pointeur char peut pointer sur un caractère isolé ou sur les éléments d'un tableau de caractères (une chaîne).

- Affectation

On peut attribuer l'adresse d'une chaîne de caractères constante:
*char *c;*
c="Ceci est une chaîne constante";
On pourra lire cette chaîne de caractères, mais il est recommandé de ne pas la modifier.

- Initialisation

Un pointeur sur char peut être initialisé lors de sa déclaration, si on lui affecte l'adresse d'une chaîne de caractères constante.

*char *b="Bonjour!!! ";*

Remarques:
- On utilise des tableaux de caractères pour déclarer des chaînes de caractères que nous voulons modifier.
- On utilise des pointeurs sur char pour manipuler des chaînes de caractères constantes (dont le contenu ne change pas).
- On utilisera de préférence des pointeurs pour effectuer des manipulations à l'intérieur des tableaux de caractères.

d) Pointeurs et tableaux à deux dimensions

L'arithmétique des pointeurs se laisse élargir avec toutes ses conséquences sur les tableaux à 2 dimensions.

Exemple:
Soit un tableau *int M[4][10]=* *{{0,1,2,3,4,5,6,7,8,9,},*
{10,11,12,13,14,15,16,17,18,19},
{20,21,22,23,24,25,26,27,28,29},
{30,31,32,33,34,35,36,37,38,39}};
Le nom du tableau M représente l'adresse du premier élément du tableau et pointe sur le tableau M[0] qui a pour valeur: {0,1,2,3,4,5,6,7,8,9,}.

L'expression M+1 est l'adresse du deuxième élément du tableau et pointe sur M[1] qui a pour valeur {10,11,12,13,14,15,16,17,18,19}.

⇒ M+i désigne l'adresse du tableau M[i].

Comment accéder à l'aide de pointeurs aux éléments de chaque composante du tableau M[0][0], …?

*int *p;*
*p=(int *)M;* Conversion forcée (obligatoire pour les pointeurs sur les tableaux à 2D)

Dû à la mémorisation ligne par ligne des tableaux 2D, on peut maintenant traiter M à l'aide de p comme un tableau unidimensionnel de dimension 4×10=40.
*(p+i) sera donc le contenu de l'élément de la ligne E(i/10) et de la colonne i%10

Exemple:
int A[3][4];
A[0][0]=1;
A[0][1]=2;
A[1][0]=10;
A[1][1]=20;
L'adresse de l'élément A[i][j] se calcule donc: A+i×4+j

Pour pouvoir travailler à l'aide de pointeurs sur un tableau à 2D, il faut connaître 4 données:
- l'adresse du premier élément du tableau converti dans le type simple des éléments du tableau (int *M).
- La longueur d'une ligne réservée en mémoire (Cf. déclaration).
- Le nombre d'éléments effectivement utilisés dans une ligne.
- Le nombre de lignes effectivement utilisées.

4) Tableaux de pointeurs

Si l'on a besoin d'un ensemble de pointeurs du même type, on peut les réunir dans un tableau de pointeurs.

- Déclaration

type *nom[N];

Exemple: *double *A[10];* Déclare un tableau de 10 pointeurs sur des rationnels de type double dont les adresses et les valeurs ne sont pas encore définies.

On utilise en général les tableaux de pointeurs pour mémoriser de façon économique des chaînes de caractères de différentes longueurs. On considèrera donc essentiellement des tableaux de pointeurs sur des chaînes de caractères.

- Initialisation

*char *jour={"lundi";"mardi";"mercredi";"jeudi";"vendredi";"samedi";"dimanche"};*

Déclare un tableau jour[] de 7 pointeurs sur char. Chacun des pointeurs est initialisé avec l'adresse de l'une des 7 chaînes de caractères.
Ainsi:
jour[i] désigne l'adresse contenue dans l'élément i de jour = l'adresse de la première composante
jour[i]+j désigne l'adresse de la j-ième composante
*(jour[i]+j) désigne le contenu de la j-ième composante

Exercice 13

5) Allocation dynamique de mémoire

a) Déclaration statique de données

Dans un programme, chaque variable a besoin d'un certain nombre d'octets en mémoire. Jusqu'ici, la réservation de la mémoire s'est déroulée automatiquement par l'emploi des déclarations des variables. Dans tous les cas, le nombre d'octets à réserver était déjà connu pendant la compilation. C'est la déclaration statique des variables.

Exemple:

int a;	réservation de 2 octets
char B[10][20];	réservation de 200 octets
float c;	réservation de 4 octets

Le nombre p d'octets à réserver pour un pointeur dépend de la machine, mais il est connu à la compilation.

*double *g;*	réservation de p octets
*char *H;*	réservation de p octets
float I[10];	réservation de 10×p octets

b) Allocation dynamique

On doit souvent travailler avec des données dont on ne peut pas prévoir le nombre et la grandeur lors de la programmation. Ce serait un gaspillage de réserver toujours l'espace prévisible. Il existe un moyen de gérer la mémoire lors de l'exécution du programme.

Exemple: *char *texte[10];*
Pour 10 pointeurs: 10×p octets. Mais on ne peut savoir à l'avance le nombre d'octets à réserver pour les phrases elles-mêmes, puisqu'elles ne seront introduites que lors de l'exécution du programme.
La réservation de la mémoire pour ces 10 phrases peut se faire seulement pendant l'exécution du programme: c'est l'allocation dynamique de la mémoire.

c) Fonction malloc et opérateur sizeof

- malloc (stdlib.h)

malloc(N); fournit l'adresse d'un bloc en mémoire de N octets libres ou 0 si il n'y a pas assez de mémoire disponible.

Exemple:
*char *T;*
T=malloc(400); Fournit l'adresse d'un bloc de 400 octets libres et l'affecte à T. S'il n'y a pas assez de mémoire, T obtient la valeur 0

Remarque: Si T=0, T ne pointe nulle part.

- sizeof (stdlib.h)

Si l'on veut réserver de la mémoire pour des données d'un type dont la grandeur varie d'une machine à l'autre, nous avons besoin de la grandeur effective d'une donnée de ce type: sizeof retourne la grandeur de son argument.

sizeof(var) Fournit la grandeur de la variable "var"
sizeof(const) Fournit la grandeur de la variable "const"
sizeof(type) Fournit la grandeur d'un objet de type "type"

Exemple 1:
sizeof(int) → 2
sizeof(double) → 8
sizeof(4,5) → 8
int A[10]
sizeof(A) → 20
char B[5][10]
sizeof(B) → 50
Exemple 2:
On veut réserver de la mémoire pour X valeurs de type int, X est lue au clavier:

```
int X;
int *p;
printf("Entrer le nombre de valeurs:");
scanf("%d",&X);
p=malloc(X*sizeof(int));
```

Chapitre X – Les fonctions

1) Variables locales, variables globales

Si une variable est utilisée uniquement de manière locale, sa déclaration se fait dans la fonction où elle est utilisée.

Exemple:

```
main()                          void fonction()
    {                               {
    int A;                          int A;
    }                               }
```

Si une variable doit être disponible pour plusieurs fonctions du programme, elle doit être déclarée de manière globale, c'est à dire juste après les include (et donc aussi avant le main).

```
#include <stdio.h>
int A;
void main()
    {
    …
    }
```

On cherchera à utiliser au minimum les variables globales car:

- elles créent des liens invisibles entre les fonctions
- elles risquent d'être cachées par des variables locales du même nom

On utilisera couramment le passage des variables comme paramètre de fonction.

2) Déclaration et définition

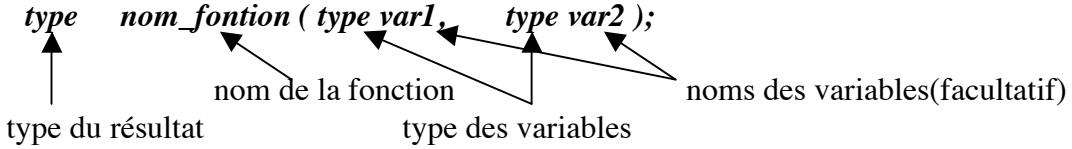
En général, le nom d'une fonction apparaît trois fois dans un programme:

1) Lors de sa déclaration
2) Lors de son appel
3) Lors de sa définition

a) Déclaration

En C, il faut déclarer une fonction avant de pouvoir l'utiliser. Cette déclaration informe le compilateur du type des paramètres et du résultat de la fonction. Seule la fonction main n'a pas besoin d'être déclarée.
Si on définit la fonction (dans le programme) avant de l'appeler, la déclaration est inutile, mais: afin de faciliter la lecture d'un programme, on conseille de définir les fonctions après le main(), il faut donc les déclarer avant (le main), juste après les include.

Syntaxe:



Déclaration locale: une fonction peut être déclarée localement dans la fonction qui l'appelle (avant les déclarations de variables), elle est alors disponible à cette fonction.
Déclaration globale: une fonction peut être déclarée globalement au début du programme (derrière les include), elle est alors disponible à toutes les fonctions du programme.

b) Définition

Syntaxe:

```
type    nom_fontion ( type var1,    type var2 )
        {
        déclarations des variables locales;
        instructions;
        }
```

Si une fonction fournit un résultat de type T, on dit que la fonction est de type T, ou a le type T.

Exemple 1:

```
int MAX (int N1, int N2)        Fonction ayant deux paramètres int, retournant un
        {                               résultat int (le max)
        if(N1>N2)    return N1;
        else    return N2;
        }
```

Exemple 2: fonction sans paramètre

```
float PI(void)
        {
        return 3.14159;
        }
```

Une fonction peut fournir comme résultat:

- un type arithmétique
- une structure, une réunion (Cf. chapitre XII)
- un pointeur
- void

Elle ne peut pas fournir des tableaux, des fonctions (mais OK pour les pointeurs)
Si une fonction n'a pas de paramètres:
Liste des paramètres= (void) ou ()
Si une fonction ne fournit pas de résultats, il faut indiquer void comme type du résultat
☝: Le type par défaut d'une fonction est int, si le type n'est pas déclaré explicitement, le résultat sera de type int.

Remarque: la fonction main est de type int: int main(void) mais on peut écrire main()

Exemple:

```
main()          Définition de main
        {
        int FA(int X, int Y);   déclaration de FA (locale à main)
        int I;
        I=FA(2,3);              appel de FA
        }
```

```
int FA(int X, int Y)          Définition de FA
        {
        int FB(int N, int M);  déclaration de FB (locale à FA)
        int J;
        J=FB(20,30);           appel de FB
        }
int FB(int N, int M)          Définition de FB
        {
        …
        }
```

Dans ce cas, la fonction FB ne peut être appelée que par FA, et FA ne peut être appelée que par main. On préférera:

```
int FA(int X, int Y);         Déclaration de FA (globale)
int FB(int N, int M);         Définition de FB (globale)
main()        Définition de main
        {
        int I;
        I=FA(2,3);            appel de FA
        }
int FA(int X, int Y)          Définition de FA
        {
        int J;
        J=FB(20,30);          appel de FB
        }
int FB(int N, int M)          Définition de FB
        {
        …
        }
```

3) Renvoyer un résultat

On utilise à la fin des instructions dans la fonction la commande:

return expression;

La commande return a pour effet:

- L'évaluation de l'expression
- La conversion automatique du résultat de expression dans le type de la fonction
- Le renvoie du résultat
- La terminaison de la fonction

Exemple 1:

```
double CARRE (double X)
        {
        return X*X;                Fournit comme résultat le carré d'un double fourni
        }                          comme paramètre
```

Exemple 2:

```
double TANGENTE (double X)
        {
        if(cos(X)!=0)
                return sin(X)/cos(X);          Fournit comme résultat le rapport sin/cos
```

```
        else                                    sinon
                printf("Erreur!\n");            affiche erreur
        }
```

Exemple 3:

```
void LIGNE (int L)
        {
        int I;
        for(I=0;I<L;I++)
                printf("*")
        printf("\n");
        }
```

Exercice 14

4) Paramètres d'une fonction

a) Passage des paramètres par valeur

En C, le passage des paramètres se fait toujours par valeur, c'est-à-dire: les fonctions n'obtiennent que les valeurs de leurs paramètres et n'ont pas accès aux variables elles-mêmes.
Les paramètres d'une fonction sont à considérer comme des variables locales qui sont initialisées automatiquement par les valeurs indiquées lors d'un appel. A l'intérieur de la fonction, on peut donc changer les valeurs des paramètres sans influencer les valeurs originales dans les fonctions appelantes.

Exemple:
La fonction ETOILE dessine une ligne de N étoiles, le paramètre N est modifié à l'intérieur de la fonction.

```
void ETOILE(int N)
        {
        while(N>0)                      En utilisant N comme compteur, on n'a pas besoin
                {                       de l'indice d'aide l comme dans la fonction LIGNE
                printf("*");            précédente
                N--;
                }
        printf("\n");
        }
```

La fonction TRIANGLE appelle la fonction ETOILE en utilisant la variable L comme paramètre.

```
void TRIANGLE (void)
        {
        int L;
        for(L=1;L<10;L++)ETOILE(L);
        }
```

Au moment de l'appel, la valeur de L est copiée dans N. La variable N peut donc être décrémentée dans ETOILE, sans influencer la valeur originale de L.

b) Passage de l'adresse d'une variable

On l'a vu précédemment, une fonction n'obtient que les valeurs de ses paramètres.

Pour changer la valeur d'une variable de la fonction appelante, nous procédons ainsi:

- La fonction appelante doit fournir l'adresse de la variable
- La fonction appelée doit déclarer le premier paramètre comme pointeur

On pourra alors atteindre la variable à l'aide du pointeur.

Exemple:
On veut écrire une fonction "PERMUT" qui échange le contenu de deux variables int. En première approche (fausse) on écrirait:

```
void PERMUT (int A, int B)
    {
    int AIDE;
    AIDE=A;
    A=B;
    B=AIDE;
    }
```

On appelle PERMUT(X, Y), résultat: X et Y restent inchangés. Pourquoi? Lors de l'appel, les valeurs de X et Y sont copiées dans les paramètres A et B. PERMUT échange bien le contenu des variables locales A et B, mais les valeurs de X et Y restent les mêmes.

APPEL			PERMUT		
X	Y		A	B	AIDE
3	4	---appel-->	3	4	
3	4	<--retour---	4	3	3

Pour pouvoir modifier le contenu de X et Y, la fonction PERMUT a besoin des adresses de X et Y. En utilisant les pointeurs, on écrit une deuxième fonction:

```
void PERMUT (int *A, int *B)
    {
    int AIDE;
    AIDE=*A;
    *A=*B;
    *B=AIDE;
    }
```

Résultat: le contenu des variables X et Y est échangé. Pourquoi? Lors de l'appel, les adressses de X et Y sont copiées dans les pointeurs A et B, PERMUT échange ensuite le contenu des adresses indiquées par les pointeurs A et B.
L'appel se fera de la manière suivante: PERMUT(&X, &Y)

APPEL			PERMUT		
X	Y		*A	*B	AIDE
3	4	---appel-->	3	4	
4	3	<--retour---	4	3	3

c) Passage de l'adresse d'un tableau à une dimension

Comme il est impossible de passer la "valeur" de tout un tableau à une fonction, on fournit l'adresse d'un élément du tableau. En général, on fournit l'adresse du premier élément du tableau: le nom du tableau.

Dans la liste des paramètres d'une fonction, on peut déclarer un tableau par le nom suivit de crochets:

type NOM[]

Ou, plus simplement par un pointeur sur le type des éléments du tableau:

type *NOM

Exemple:
La fonction strlen calcule et retourne la longueur d'une chaîne de caractères fournie comme paramètre.

```
int strlen(char *s)      ← on aurait pu écrire char s[]
        {
        int N;
        for(N=0;*s!='\0',s++)
                N++;
        return N;
        }
```

On utilisera par la suite cette première notation, afin de mettre en évidence que le paramètre est un pointeur variable que l'on peut modifier à l'intérieur de la fonction.

Exercice 15 (Remarque: on voit qu'il est possible de fournir une partie d'un tableau à une fonction)
: pour qu'une fonction puisse travailler correctement avec un tableau, il est prudent de lui fournir la dimension du tableau, ou le nombre d'éléments à traiter, sinon la fonction risque de sortir du tableau.

d) Passage de l'adresse d'un tableau à deux dimensions

Le plus simple est de passer l'adresse du premier élément du tableau:&tab[0][0], ou d'utiliser le cast: (float *)tab.

Exemple: On cherche à faire une fonction qui calcule la somme des éléments d'un tableau à deux dimensions dont on fournit les dimensions N et M comme paramètres.
Comment pouvons-nous passer l'adresse de la matrice à la fonction ?
Par analogie avec ce que nous avons vu précédemment, nous pourrions envisager de déclarer le tableau concerné dans l'en-tête de la fonction sous la forme A[][] . Dans le cas d'un tableau à deux dimensions, cette méthode ne fournit pas assez de données, parce que le compilateur a besoin de la deuxième dimension du tableau pour déterminer l'adresse d'un élément A[i][j].
Une solution praticable consiste à faire en sorte que la fonction reçoive un pointeur (de type float*) sur le début de la matrice et de parcourir tous les éléments comme s'il s'agissait d'un tableau à une dimension N*M.

Cela nous conduit à cette fonction:

```
float SOMME(float *A, int N, int M)
{
int I;
float S;
for (I=0; I<N*M; I++)
S += A[I];
return S;
}
```

Lors d'un appel de cette fonction, la seule difficulté consiste à transmettre l'adresse du début du tableau sous forme d'un pointeur sur float. Prenons par exemple un tableau déclaré par:
float A[3][4];
Le nom A correspond à la bonne adresse, mais cette adresse est du type "pointeur sur un tableau de 4 éléments de type float". Si notre fonction est correctement déclarée, le compilateur la convertira automatiquement dans une adresse du type "pointeur sur float". Toutefois, comme nous l'avons déjà remarqué au chapitre IX, on gagne en lisibilité et l'on évite d'éventuels messages d'avertissement si on utilise l'opérateur de conversion forcée (cast).

Solution:
Voici finalement un programme faisant appel à notre fonction SOMME:

```
#include <stdio.h>
float SOMME(float A, int N, int M);       / déclaration de la fonction SOMME     */
main()
{
float T[3][4] = {{1, 2, 3, 4},{5, 6, 7, 8},{9,10,11,12}};
printf("Somme des éléments : %f\n",SOMME((float*)T, 3, 4) );
}
```

Rappel: Rappelons encore une fois que lors de l'interprétation d'un tableau à deux dimensions comme tableau unidimensionnel, il faut calculer les adresses des composantes à l'aide du nombre de colonnes maximal réservé lors de la déclaration.

Chapitre XI – Les fichiers séquentiels

En C, les communications d'un programme avec son environnement se font par l'intermédiaire de fichiers. Pour le programmateur, tous les périphériques, y compris le clavier et l'écran, sont des fichiers. Jusqu'ici, on a lu des données dans le fichier d'entrée standard (le clavier) et on les a écrits dans le fichier de sortie standard (l'écran).
On va voir dans ce chapitre que l'on peut créer, lire et modifier nous même des fichiers sur les périphériques disponibles.

1) Définition

Un fichier (en anglais: file) est un ensemble de données stockées en général sur un support externe (disque dur, disquette, CD, bande magnétique, …). Un fichier structuré contient une suite d'enregistrements homogènes, qui regroupent le plus souvent plusieurs composantes appartenant à un ensemble (champs).
Dans les fichiers séquentiels, les enregistrements sont mémorisés consécutivement dans l'ordre de leur entrée et peuvent seulement être lus dans cet ordre. Si on a besoin d'un enregistrement précis, il faut lire tous les enregistrements qui le précèdent, en commençant par le premier.

Les fichiers séquentiels que nous allons considérer auront les propriétés suivantes:

- les fichiers se trouvent soit en état d'écriture, soit en état de lecture, on ne pourra pas simultanément lire et écrire dans un fichier.
- à un moment donné, on peut uniquement accéder à un seul enregistrement: celui qui se trouve en face de la tête de lecture.
- après chaque accès, la tête de lecture/écriture est déplacée derrière la donnée lue/écrite en dernier lieu.

2) La mémoire tampon

Pour des raisons d'efficacité, les accès à un fichier se font par l'intermédiaire d'une mémoire tampon (en anglais: buffer); La mémoire tampon est une zone de la mémoire centrale de la machine réservée à un ou plusieurs enregistrements du fichier. L'utilisation de la mémoire tampon a l'effet de réduire le nombre d'accès à la périphérie d'une part et le nombre des mouvements de la tête de lecture/écriture d'autre part.

3) L'accès aux fichiers séquentiels

Avant de lire ou d'écrire dans un fichier, l'accès est notifié par *fopen*. *fopen* accepte le nom d'un fichier, négocie avec le système d'exploitation et fournit un pointeur spécifique qui sera ensuite utilisé lors de l'écriture/lecture du fichier. Après traitement, il faut annuler la liaison entre le nom du fichier et le pointeur à l'aide de la commande *fclose*.
On peut dire de manière plus simple qu'entre *fopen* et *fclose* le fichier est ouvert.

a) Le type *FILE

Pour pouvoir travailler avec un fichier, un programme a besoin d'un certain nombre d'informations au sujet de ce fichier:

- l'adresse de la mémoire tampon

- la position actuelle de la tête de lecture/écriture
- le type d'accès au fichier: lecture ou écriture
- l'état d'erreur
- …

On n'aura pas à s'occuper de ces informations, elles sont rassemblées dans une structure de type spécifique FILE. Lorsque l'on ouvre un fichier avec la commande fopen, le système génère automatiquement un bloc du type FILE et nous fournit son adresse.

Tout ce que l'on a à faire est:

- déclarer un pointeur de type **FILE* pour chaque fichier dont on a besoin (ex*: FILE *fich;*)
- affecter l'adresse retournée par *fopen* à ce pointeur
- employer le pointeur à la place du nom du fichier dans toutes les instructions de lecture/écriture
- libérer le pointeur à la fin du traitement à l'aide de *fclose*

b) fopen: ouverture

fopen affecte au pointeur de type FILE l'adresse du fichier.

fich=fopen(nom_de_fichier, mode);

- nom_de_fichier est une chaîne de caractères, soit "fichier.dat", soit chaine[20], si chaine[20] a été déclarée de type char et initialisée
- mode est le mode de lecture/écriture
  - "w" (write): mode écriture (si le fichier n'existe pas, la fonction le crée, si le fichier existe, la fonction le vide)
  - "r" (read): mode lecture (si le fichier n'existe pas, la fonction ne le crée pas)
  - "a" (ajout): mode ajout écriture (si le fichier n'existe pas, la fonction le crée, si le fichier existe, les écriture auront lieu à la fin du fichier)

Le type d'ouverture peut être agrémenté de deux caractères:

- "b": le fichier est considéré en mode binaire. Il peut donc contenir des données qui sont transférées sans interprétation par les fonctions de la bibliothèque
- "+": le fichier est ouvert dans le mode complémentaire au mode de base

Exemple: "r+" le fichier est ouvert en mode lecture et plus, c'est-à-dire lecture + écriture

Si le système ne peut pas ouvrir le fichier, il retourne un pointeur nul (0)
Exemple:
fich1=fopen(fichier.dat","r");
iIf(fich1==0)printf("Le fichier n'a pas été ouvert\n");

c) Les instructions de lecture et d'écriture

Elles sont analogues à putchar, getchar, printf et scanf. MAIS on ajoute un "f" avant et l'on ajoute en argument le nom du pointeur de type *FILE.

- Ecriture d'un caractère

fputc('a',fich) Ecrira le caractère a dans le fichier pointé par fich

- Lecture d'un caractère

lettre=fgetc(fich) lit un caractère dans le fichier pointé par fich et le "met" dans lettre

- Ecriture d'une information

fprintf("fich,"la valeur est %f\n",expr1) Ecrira dans le fichier pointé par fich le texte "la valeur est ", suivi de la valeur de expr1 au format float et d'un retour à la ligne

- Lecture d'une information

fscanf("fich," %d",&n) Lira dans le fichier pointé par fich la valeur d'un int qui sera stocké à l'adresse de n (&n)

d) Détection de la fin d'un fichier

feof(fich) retourne 0 si la tête de lecture référencée par fich n'est pas à la fin du fichier, retourne 1 si elle est à la fin

Exemple: boucle typique pour lire les enregistrements

```
while(!feof(fich))
        {
        fscanf(fich,"%s",NOM);
        }
```

Résumé sur les fichiers:

Déclaration du pointeur FILE	***FILE *fp;***
Ouverture en écriture	***fp=fopen(NOM,"w");***
Ouverture en lecture	***fp=fopen(NOM,"r");***
Fermeture	***fclose(fp);***
Fin de fichier	***feof (fp);***
Ecriture	***fprintf("fp,"…",variable);***
	fputc(char,fp);
Lecture	***fscanf("fp,"…",&variable);***
	c=fgetc(fp);

Exemple: créer et afficher un fichier séquentiel, ***exercice 16***

e) Déplacement dans un fichier

- **fseek(fich, deplacement, position)**

- fich pointe sur le fichier dans lequel on veut se déplacer
- deplacement est le déplacement que l'on veut effectuer (c'est un entier long)
- position est le point de départ du déplacement (c'est un entier int), il peut prendre 3 valeurs:
  - *SEEK_SET*: déplacement relatif au début du fichier
  - *SEEK_END*: déplacement relatif à la fin du fichier
  - *SEEK_CUR*: déplacement relatif à la position courante

- **ftell (fich)**

retourne la valeur de la position courante dans le fichier (pour les fichiers binaires, ce sera le nombre d'octets entre la position courante et le début du fichier). La valeur retournée est un entier de type long.

Chapitre XII – Les "plus"

Vous trouverez dans ce chapitre, tout ce que je n'ai pas réussi à "caser" ailleurs, mais attention, ce chapitre n'est pas une "poubelle", voyez-le plutôt comme un "fourre-tout", qui se révélera utile pour passer au C++…

1) La structure de contrôle switch

Le switch est une"table de branchement" qui permet d'éviter les imbrications de if.

Syntaxe:

```
switch(expression)
    {
    case val1:      instruction1;
                    break;
    case val2:      instruction2;
                    break;
    …
    default:        instructionX;
                    break;
    }
```

- Expression est évaluée comme une valeur entière.
- Les valeurs des cases sont évaluées comme des constantes entières, l'exécution se fait à partir du case dont la valeur correspond à expression. Elle s'exécute en séquence jusqu'à la rencontre d'une instruction break.
- Les instructions qui suivent la condition default sont exécutées lorsque aucune constante des case n'est égale à la valeur retournée par expression.
- L'ordre des case et du default n'a pas d'importance.
- L'exécution à partir d'un case continue sur les instructions des autres case tant qu'un break n'est pas rencontré.
- Plusieurs valeurs de case peuvent aboutir sur les mêmes instructions
- Le dernier break est facultatif

Exercice 17

2) Les ruptures de séquences

a) continue

Le continue est utilisé avec les boucles, il provoque le passage à l'itération suivante de la boucle en sautant la fin du bloc. Il provoque la non-exécution des instructions qui le suivent à l'intérieur du bloc.

Exemple:

```
for(i=0,j=0;(c=getchar()!=EOF);i++)      /* EOF est un caractère prédéfini marquant la */
        {                                                   /* fin de la saisie */
        if((c==' ')||(c=='\t')||(c=='\n'))continue;
        j++;
        }
```

On compte le nombre de caractères non-blancs entrés au clavier (blanc: espace, tabulation, saut de ligne).
A la fin de l'exécution: i contient le nombre total de caractères entrés au clavier, j contient le nombre de caractères non blanc.

b) break

Le break a déjà été vu dans le switch. Plus généralement, il permet de sortir de la boucle d'itération (on ne sort que d'un niveau de boucle).
Exemple:

```
for(i=0,j=0;(c=getchar()!=EOF);i++)
        {
        if(c=='\n')break;
        if((c==' ') continue;
        if(c=='\t') continue;
        j++;
        }
```

On compte le nombre de caractères non-blancs entrés au clavier (blanc: espace, tabulation) jusqu'au retour à la ligne. Lorsque celui-ci est rencontré, le break provoque la sortie de la boucle for (idem avec un while)

c) goto

Le goto permet d'aller n'importe où à l'intérieur d'une fonction (donc du main également). Son utilisation systématique nuit à la lisibilité des programmes.
Exemple:

```
        goto erreur;
        …
        erreur: printf("erreur!!!\n");
```

3) Les types complexes

a) Les structures

Une structure est un objet composé de plusieurs champs qui sert à représenter un objet réel ou un concept. Par exemple, structure: voiture, champs: marque, modèle, couleur, année, …

Syntaxe:

```
struct nom_de_structure
        {
        type 1 nom_champ1;
        type 2 nom_champ2;
        type 3 nom_champ3;
        …
        }variables;
```

Exemple:

```
struct date
        {
        int jour;
```

```
int mois;
int annee;
}
```

La définition d'une structure ne réserve pas d'espace mémoire, il faut définir des variables correspondant à ce modèle de structure:

- Variable "simple"

Soit, après la définition de date

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

La structure obdate a trois composantes:

obdate.jour	Que l'on peut initialiser:
obdate.mois	*obdate.jour=24;*
obdate.annee	*obdate.mois=11;*

Soit, avec la définition de date

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

- Variable "pointeur"

Soit, après la définition de date

```
struct date *ptdate;
```

Soit, avec la définition de date

```
struct date
        {
        int jour;
        int mois;
        int annee;
        }*ptdate;
```

Initialisation:

```
ptdate->jour=1;          ou      (*ptdate).jour=1;
ptdate->mois=1;          ou      (*ptdate).mois=1;
ptdate->annee=2006;          ou      (*ptdate).annee=2006;
```

Le nom de la structure est optionnel (elle est alors anonyme), si l'on met des variables. On ne peut pas le référencer par la suite.
On autorise les opérations suivantes sur les structures:

- affectation d'une variable structurée par une autre variable structurée du même type:
  oddate2=obdate;
- test d'égalité ou d'inégalité entre deux variables structurées du même type:
  oddate2==obdate;
  oddate2!=obdate;
- le retour d'une variable structurée par une fonction:
  oddate=fonction();
- le passage en argument d'une variable structurée à une fonction:
  resul=fonction(obdate);

b) Les unions

Les unions permettent l'utilisation d'un même espace-mémoire par des données de types différents à des moments différents.

Syntaxe:
union nom_de_union
{
type 1 nom_champ1;
type 2 nom_champ2;
type 3 nom_champ3;
…
}variables;

Exemple:
union zone
{
int entier;
long ent_lg;
float flottant;
}z1,z2;

Cet exemple défini deux variables z1 et z2 construites sur le modèle d'une zone qui peut contenir soit un int, soit un long, soit un float.
Lorsque l'on définit une variable correspondant à un type union, le compilateur réserve l'espace-mémoire nécessaire pour stocker le plus grand des champs appartenant à l'union (dans l'exemple: float).
L'accès aux champs se fait comme pour la structure.
Une union ne contient cependant qu'une donnée à la fois, l'accès à un champ de l'union pour obtenir une valeur doit être fait dans le même type qui a été utilisé pour la stocker.
Les différents champs d'une union ont la même adresse: &z1.entier=&z1.ent_lg=&z1.flottant
☝: Les unions ne sont pas utilisées pour faire des conversions. Elles ont été inventées pour utiliser un même espace mémoire avec des types de données différents dans des étapes différentes d'un même programme (en conclusion, pour utiliser un minimum d'espace mémoire, ce qui n'est plus très utile de nos jours…).

c) Les énumérations

Les énumérations servent à offrir des possibilités de gestion de constantes numériques.

Syntaxe:
Enum nom_de_enumeration
{
enumeration1;
enumeration 2;
…
}variables;

Les valeurs associées aux différentes constantes symboliques sont, par défaut, définies ainsi:
- la première constante est associée à la valeur 0
- les constantes suivantes suivent une progression de 1

On peut cependant fixer une valeur à chaque énumérateur:
- en faisant suivre l'énumérateur du signe = et de la valeur entière exprimée par une constante
- la progression reprend à 1 pour l'énumérateur suivant

Exemple 1:
Enum couleurs
{
rouge, vert, bleu;
}rvb;
On définit ainsi les constantes symboliques:
rouge correspond à 0
vert correspond à 1
bleu correspond à 2

Exemple 2:
Enum autres_couleurs
{
violet=4,
orange,
jaune=8;
}voj;
On définit ainsi les constantes symboliques:
violet correspond à 4
orange correspond à 5
jaune correspond à 8

4) Définition de types

Il est possible, grâce au déclarateur **typedef** de définir un type nouveau.

Syntaxe:
typedef type nom_de_type;

Exemple 1: *typedef int entier;*
Cette ligne définit un type synonyme appelé entier ayant les mêmes caractéristiques que le type prédéfini int. Une fois cette définition réalisée, on peut utiliser ce nouveau type pour définir des variables.
Exemple 2: *typedef int tab[10];*
tab tt;
Ici, tab devient un type synonyme correspondant au tableau de 10 int. La variable tt est alors un tableau de 10 int.
typedef ne réserve pas de mémoire, le nom est un type, il est donc inaccessible comme variable.
Exemple 3:
typedef struct
{
int jour;
int mois;
int annee;
}date;
date obdate;
Ainsi, obdate est un objet correspondant au type date, c'est-à-dire une structure de 3 int.

5) Les instructions du précompilateur

Ces instructions se placent au tout début du programme, avant le main, et même avant les éventuelles déclarations de fonctions.

a) include

Nous l'avons déjà vu au tout début de ce cours, il permet l'inclusion de fichier:

#include <nom_de_fichier>

b) Définition de constantes

#define NOM valeur

Exemple 1: #define pi 3.1416
Exemple 2: #define N 20

On pourra ensuite utiliser N qui est initialisé à la valeur 20, on ne peut plus changer sa valeur.

c) Définition de macro-expressions

Exemple 1: #define m(x) (128*(x)+342*(x)*(x))
Exemple 2: #define add(x1,x2) ((x1)+=(x2))

: mettre des parenthèses autour de la variable, même dans l'expression de la macro.