# Langage C : notes du cours

Hanna Klaudel, Université d'Evry

---

## Plan du cours :

1. Introduction (notion d'algorithme, programme, fonctionnement interne de l'ordinateur, interpréteur de commandes shell, quelques commandes de base).
2. Préliminaires (notion de type simple, variable, constante, affectation, expression d'algorithme en pseudo-langage et codage, fonctions de base d'I/O).
3. Expressions et instructions (expressions arithmétiques, Booléennes, opérateurs associés ; instructions itératives, conditionnelles, autres).
4. Types de données (tableaux, chaînes de caractères, structures, unions).
5. Structure d'un programme complexe (notion de bloc, portée des variables).
6. Fonctions (passage de paramètres, valeur de retour, variables locales, variables globales).
7. Récursivité.

## Bibliographie

- B. Kernighan et D. Ritchie. *Le langage C*. Dunod, 2002.
- C. Delannoy. *Programmer en Langage C*. Eyrolles. 2002.

## Notions de base

- L'informatique est une science qui s'occupe du traitement automatisé de l'information (à l'aide des logiciels ou programmes).
- Un programme peut être assimilé à une recette ou à une procédure à appliquer pour réaliser quelque chose.

1. (cuisine) Pour faire un kir on a besoin de rassembler les ingrédients (du vin blanc et de la crème de cassis) et de les mélanger dans un verre en respectant les proportions ;

2. (calcul) Pour savoir quels élèves avaient la moyenne du bac supérieure à la moyenne de la classe, on a besoin de connaître la moyenne de chaque élève (éventuellement, on a besoin de la calculer), la moyenne de la classe, le nombre d'élèves dans la classe, et enfin, on doit comparer la moyenne de chacun à la moyenne de la classe et choisir comment on va présenter le résultat (sous forme d'une liste, d'un tableau etc).

# Conception d'un programme

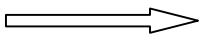
Deux phases :

- Algorithmique : la modélisation et abstraction du problème qui mène à l'élaboration d'une solution automatisée :
  - Choix des structures de données (tableau, liste, etc)
  - Conception des algorithmes (suite d'opérations à faire)

  Al Khowarizmi, Bagdad IXième siècle.

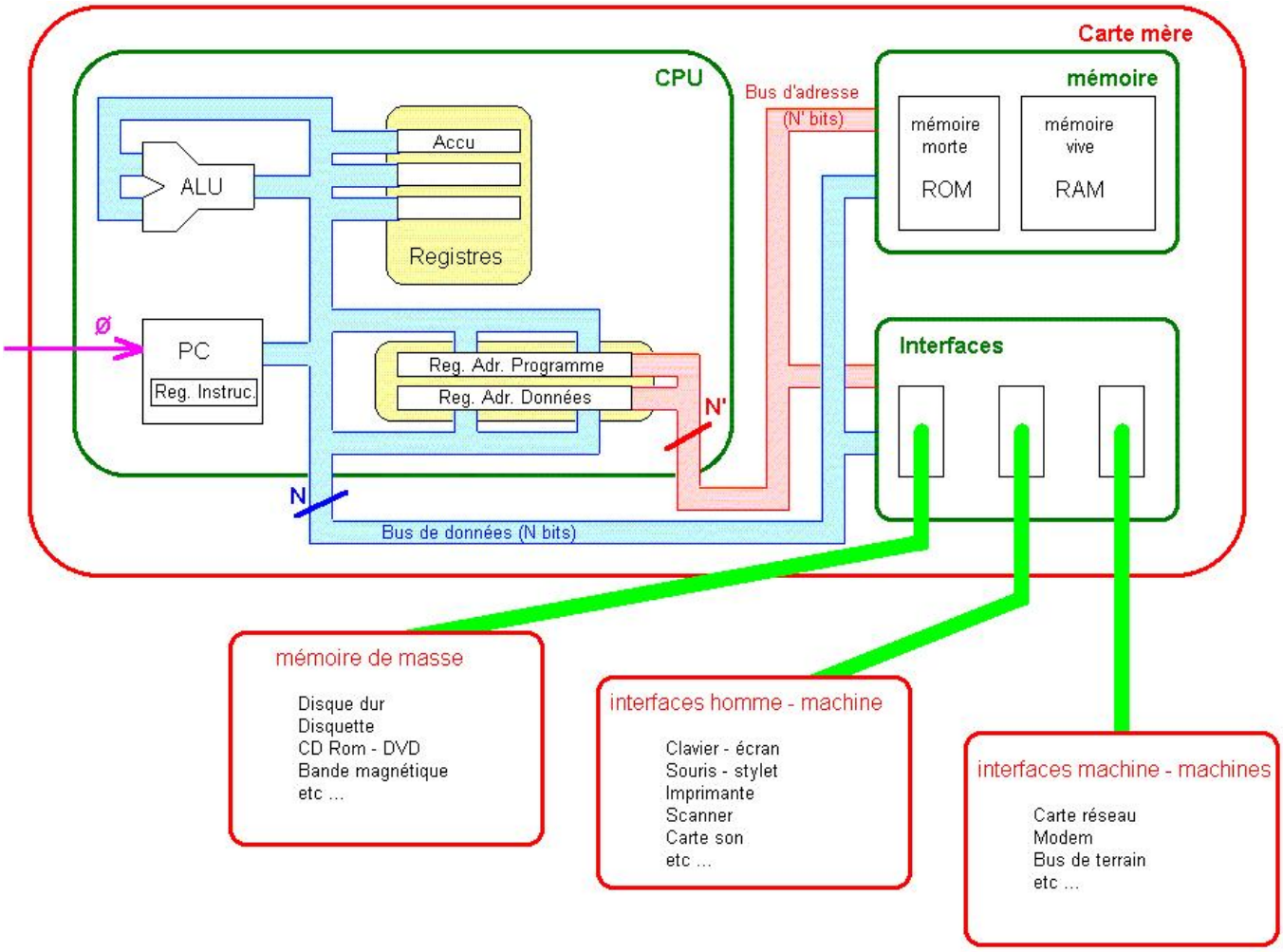
  *Encyclopedia Universalis* : « Spécification d'un schéma de calcul sous forme d'une suite finie d'opérations élémentaires obéissant à un enchaînement déterminé. »

- Programmation
  - Traduction dans un langage compréhensible par l'ordinateur des structures de données et algorithmes



ordinateur

# Fonctionnement de l'ordinateur : Modèle de von Neumann




CPU : Central Processing Unit (unité centrale de traitement) à processeur.

ALU : Unité Arithmétique et Logique : le composant qui sait faire les calculs.

Registres : mémoires internes du processeur. Un registre est capable de stocker N 0 ou 1, par exemple le résultat de la dernière opération de l'ALU. Dans un processeur, le nombre de registres est très limité (3 à 8). L'un de ces registres est plus important que les autres à accumulateur (accu).

PC (Partie Commande) : la PC commande le processeur.

## Comment commande-t-on la PC ?

- Chaque action que PC sait faire est numérotée (en binaire)

  à code opération ou code-op.

- Le choix de la codification est fait par le fabriquant du processeur à langage machine.

- On commande l'ordinateur à l'aide d'un programme

  à suite de code-ops (stocké en mémoire).

- C'est le registre d'adresse de programme (RAP) qui sait à tout moment où on en est dans le programme.

- Le CPU va y lire l'instruction suivante, la mémorise dans le registre d'instruction (RI) et l'effectue, ajoute 1 au RAP (sauf si c'était un goto) et recommence, indéfiniment.

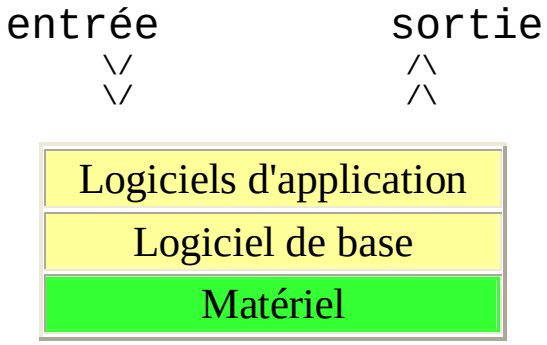
- Le registre d'adresse de données (RAD) sert à stoker l'adresse de la prochaine donnée à lire ou écrire en mémoire.

## Programmation : les différents niveaux d'abstraction.

- Matériel (Hardware) : logique numérique, microprogrammes
- Logiciel (Software) :
  - Langage machine : ajouter 2 entiers, déplacer une donnée, ...
  - Langage assembleur : représentation symbolique des instructions et données
  - Langage de programmation de haut niveau : C, C++, Fortran, Java, Lisp, ...
  - Générateurs d'application

## Logiciels de base et logiciels d'application

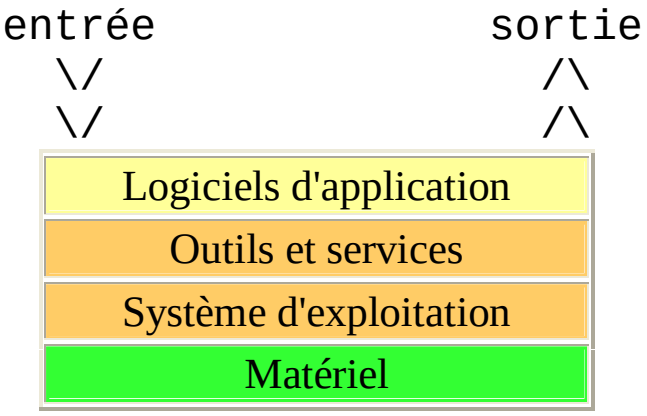
La partie logicielle (software) de l'ordinateur est subdivisée en deux parties :




- Les logiciels d'application sont les logiciels pour lesquels on emploie l'ordinateur comme le traitement de texte, la comptabilité, les bases de données, le tableur, le logiciel de dessin, etc..
- Les autres programmes définissent ce qu'on appelle le logiciel de base. On y trouve des programmes gérant les fonctions du système.

## Logiciel de base

La couche appelée logiciel de base peut être décomposée en deux sous-niveaux :




- outils et services : composés d'utilitaires, d'éditeurs (Emacs...), de compilateurs (gcc, ...);
- système d'exploitation (*operating system - OS : MS-DOS, WINDOWS, OS/2, Unix, etc*) : un ensemble de programmes qui servent d'interface entre les logiciels d'application et la machine physique (environnement), et contrôlent le fonctionnement général de l'ordinateur.
  - la gestion des fichiers et la manière de les nommer ;
  - la structuration des supports informatiques (disques, disquettes…).

## Organisation des fichiers : système de fichiers

Les fichiers sont rangés dans une structure arborescente dont la racine est /.

… comme une adresse postale lue à l'envers: Pays, Ville, Rue, Numéro de rue...

| Exemple d'arborescence | Exemples de chemins |
|---|---|
| / | / |
| `--France | /France |
| \|--Evry | /France/Evry |
| \| \|--Université | /France/Evry/toto |
| \| `--INT | ... |
| `--Paris | /France/Paris |

Un chemin peut-être absolu ou relatif. …si vous êtes déjà dans la bonne rue, le numéro vous suffit.

## Interpréteur de commandes: SHELL

Lorsque vous ouvrez une *session* sous UNIX, vous avez en général une console ou une fenêtre dans laquelle vous pouvez tapez des commandes.

Le programme avec lequel vous interagissez est le *shell*. Il va interpréter vos instructions au fur et à mesure que vous terminez de les taper. Vous lui indiquez qu'il peut commencer à interpréter à chaque fois que vous validez une introduction ou une série d'instructions en appuyant sur la touche enter.

Voir :

http://www.univ-reims.fr/Labos/LSSB/links/cea2002/web/index.html

## Prise en main

Le shell vous indique qu'il est prêt en affichant un prompt, en général le caractère dollar $ ou supérieur >, en début de ligne. Comme ceci:

```
bash$

csh>
```

Les commandes, celles que vous demandez à l'ordinateur d'exécuter sont indiquées comme ceci:

```
bash$ commande
```

Ce que celui-ci vous répond est indiqué

```
comme ceci
```

## Qui suis-je?

Commençons par une commande simple qui vous renvoie votre login.

```
bash$ whoami

tru
```

## Sur quelle machine suis-je?

hostname vous renvoie le nom de la machine sur lequel votre shell s'exécute.

```
bash$ hostname
nemo07
```

## Stop!

Ctrl-c c'est-à-dire: [ctrl] enfoncée puis lettre c ne signifie pas copier! Mais plutôt interrompre la commande actuelle.

## Bye bye

exit permet de quitter le shell.

```
bash$ exit
```

## Arguments et options des commandes

```
bash$ commande option1 ... optionN argument1  ... argumentM
```

Les `options` sont souvent de la forme -une_seule_lettre.

Les `options` modifient le comportement de votre commande alors que les *`arguments`* sont les *objets* sur lesquels la commande s'applique.

Par exemple, la commande ls :

`ls` : affiche le contenu du répertoire courant

`ls -l` : affiche le contenu du répertoire courant avec détails

`ls –la` : affiche en plus les fichiers « cachés » *commençant* par le caractère . (point).

## Commandes avec options

Dans un shell, les *expressions régulières (REgular EXpression)* forment un moyen condensé de décrire une chaîne de caractères. Pratiquement cela donne:

- * signifie *n'importe quelle chaîne de caractères dont le premier terme est alphanumérique (majuscules/minuscules comprises)*.
- A* signifie *n'importe quelle chaîne de caractères commençant par la lettre A*.
- ? signifie *exactement un seul caractère alphanumérique*
- [Aa9] signifie *soit la lettre "A majuscule", soit la lettre "a minuscule" soit le chiffre 9*.

```
bash$ ls *.html *.txt

bash$ ls -l /tmp/*A*

bash$ ls .???*
```

## Quelle est la taille qu'occupent mes données

du pour *d*isk*u*sage donne la taille de *tous* les arguments qui lui sont passés et de façon récursive.

Les options les plus fréquemment utilisées sont `-s` pour n'afficher que la somme pour chacun des arguments, `-k` pour l'exprimer en kilo octets et `-h` pour adapter le coefficient à la taille (kilo, méga, giga octets).

```
bash$ du $HOME
bash$ du -k $HOME
bash$ du -s $HOME
bash$ du -sk $HOME
bash$ du -h $HOME
```

# Manipulation des fichiers et répertoires

## Où suis-je ?

pwd *P*rint*W*orking *D*irectory

```
bash$ pwd

/home/Bis/tru/perso/cours/cea
```

## . et ..

Le . (point) représente le répertoire *courant* : c'est le répertoire dans lequel vous êtes actuellement. Le .. (point point) représente le répertoire *parent* : c'est le répertoire juste au dessus.

```
bash$ pwd

/home/Bis/tru/perso/cours/cea
```

. est ici /home/Bis/tru/perso/cours/cea

.. est ici /home/Bis/tru/perso/cours

## Se déplacer

cd *C*hange*D*irectory

```
bash$ cd
```

cd sans argument, vous ramène *chez vous* (dans $HOME).

cd peut prendre un argument, c'est le répertoire de *destination*.

```
bash$ cd ..

  bash$ cd /usr/local/bin

  bash$ cd /

  bash$ cd ../usr
```

## Lister

ls pour *L*i*S*t est une commande que nous avons vu précédemment.

```
bash$ ls [-la] fichier(s) ou répertoire(s)
```

## Renommer et déplacer un fichier ou un répertoire

mv pour *M*o*V*e

```
bash$ mv fichier1 fichier2 repertoire1 destination

bash$ mv ancien_nom nouveau_nom

bash$ mv ../fichier1 /tmp/fichier2 .
```

S'il y a plusieurs objets à déplacer, la destination est nécessairement un répertoire.

S'il n'y a que deux arguments, mv ne fait que *renommer* les deux objets de même type.

## Copier

cp pour *C*o*P*y

```
bash$ cp fichier1 fichier1.copie

bash$ cp fichier1 fichier2 *.txt repertoire

bash$ cp -r repertoire1 repertoire2
```

## Effacer un fichier

rm pour *R*emo*V*e permet d'effacer

```
bash$ rm *.bak

  bash$ rm -r repertoire

  bash$ rm -fr repertoire fichiers
```

Il N'existe AUCUN moyen d'annuler un rm -rf (rm *récursif sans confirmation*). Une fois effacé, votre fichier/répertoire est complètement *PERDU*.

## Effacer un répertoire (vide)

rmdir *R*e*M*ove *DIR*ectory

```
bash$ rmdir repertoire_vide
```

## Créer un répertoire

mkdir pour *M*a*K*e *DIR*irectory

```
bash$ mkdir nouveau_rep autre_rep
```

## Afficher un fichier

cat permet d'afficher le contenu d'un fichier sur STDOUT (en général l'écran).

```
bash$ cat /etc/resolv.conf

nameserver 132.166.192.6
nameserver 132.166.192.7
```

Si le fichier est très long, ce n'est pas pratique du tout! Mieux vaut faire défiler page par page.

## Afficher page par page

more est ce que l'on nomme un *PAGER*. Il permet d'afficher un fichier *page par page*.

- SPACE pour faire défiler en avant d'une page
- b pour faire défiler en arrière d'une page.

```
bash$ more /usr/local/bioinfo/tru/html/lundi.html
```

## Processus

## ps

ps pour *P*rocess*S*tatus permet de lister les processus en cours.

Selon les UNIX, ce sera la première ou la seconde:

```
bash$ ps -ax

bash$ ps -ef
```

## kill

```
bash$ kill 3456
```

kill termine/tue le processus dont le numéro est passé en argument.

# Introduction à la programmation en C

## Généralités

Historique : créé par Dennis Ritchie en 1972 aux labos Bell

Caractéristiques :

- langage de haut niveau, impératif, orienté bloc
- programmation système Unix
- modulaire : nombreuses librairies standard (string, math)

## Premier exemple

Imprimer les mots : Hello World, programme `hello.c` en C

```
#include <stdio.h>
main()
{
   printf("Hello World\n");
}
```

- le texte du programme est écrit à l'aide d'un éditeur (Emacs) dans un fichier dont le nom doit finir par .c
- `#include` permet d'inclure des fichiers/bibliothèques
- `main()` est une fonction sans arguments qui indique l'endroit par lequel l'exécution du programme va commencer
- les accolades `{ }` contiennent les instructions qui composent la fonction
- `printf("Hello World\n");` correspond à l'appel de la fonction standard `printf` avec l'argument `"Hello World\n"`
- les guillemets `"  "` délimitent une chaîne de caractères
- \n est le caractère d'interligne ( `\t` - tabulation, `\b` - backspace, `\"` - ", `\\` - \)

## Compilation et exécution sous Unix

Commandes Unix :

| | |
|---|---|
| `> gcc hello.c -o hello`<br>`> hello`<br>`Hello World`<br>`>` | `> gcc hello.c`<br>`> a.out (ou bien ./a.out)`<br>`Hello World`<br>`>` |

Compilation en 2 étapes :

1. *Compilation* : `source.c` -------> `objet.o`
2. *Edition de liens* : `objet.o` --------> `executable`

`hello.c` ------> `hello.o` -------> `hello`

## Un autre exemple

```
/* permet de construire des mots et des phrases */

#include <stdio.h>

void f1()
{
   printf("a");
}

void f2()
{
   printf("un");
}

void f3()
{
   printf("on");
}

void f4()
{
   printf("chat");
}

void f5()
{
   printf(" ");
}

main()
{     f2();
      f5();
      f4();
      f5();
      f3();
}
```

## Structure d'un programme

*inclusion des fichiers d'entête*

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

*définitions de fonctions*

```
void f2(){ ... }
int somme(...) { ... }
double moyenne(...) { ... }
```

*programme principal*

```
int main() { ...  }
```

- un programme est une liste de fonctions
- lors de son appel, les instructions d'une fonction sont exécutées *séquentiellement*.
- chaque fonction est indépendante des autres (variables utilisées)
- les fonctions peuvent s'appeler entre elles : dans `main()` appel de `f1()`
- la fonction `main()` est le *point d'entrée* du programme

## Définitions de variables

Une variable est identifiée par son nom et correspond à une zone en mémoire, repérée par son adresse.
Le langage C est typé. Chaque type correspond à une taille de zone mémoire (nombre d'octets).

| Types simples : | Exemples de valeurs valides : |
|---|---|
| `int /* entier */` | `2 4 -8` |
| `char /* caractère */` | `'a' 'P' '\n'` |
| `float double /* virgule flottante */` | `3.2 -1.685E24` |
| `void /* vide */` | `RIEN` |

Déclarations de variables :

```
Type NomVariable [= Expression] ;
```

```
int i;
int ratio = 6;
int j, k;

double p, p=3.0+18;

double r = somme(18, 7.0) + ratio;

char a ='a', caractere;
```

## Utilisation de variables

```
int i,j;      /* declaration de variables
                 entières i et j */
```

Toute variable doit être déclarée avant d'être utilisée.

Le nom de la variable est intimement lié à sa valeur.

La valeur de la variable varie généralement pendant l'exécution du programme.

Pour "donner une valeur" à une variable, on réalise une *instruction d'affectation* :

```
NomVariable = Expression ;
```

```
i = 8;     /* affectation à i de la valeur 8 */

j = i + 3;   /* affectation à j de la valeur
                de i (qui est actuellement 8)
                augmentée de 3 */

j = j + 1;   /* affectation à j de la valeur
                de j d'avant l'opération
                (c'est-à-dire 11) augmentée
                de 1 */
```

Dans la pratique, une variable est un couple (*adresse*, *valeur*), où *adresse* désigne la zone mémoire où est stockée la variable.

Le nombre d'octets utilisés pour le stockage dépend du type de la variable : ex. `sizeof(int)` vaut 4 octets.

## Un exemple plus grand

Programme qui imprime la table de conversion des euros en devises avec le cours = 2.11, pour 0, 10, 20, ..., 200. Donc, la table :

```
  0       0.0
 10      21.1
 20      42.2

  .........

200     422.0
```

D'abord en pseudo-langage

```
variables : euro, devise : réels

   euro <- 0
   tant que  euro <= 200  faire
       début
          devise <- euro * 2.11,
          imprimer(euro, devise),
          euro <- euro + 10
       fin
```

La valeur de la variable euro est initialement 0 et augmente de 10 à chaque exécution du corps de la boucle.

Attention, l'indentation est importante pour mieux voir la structure de l'algorithme.

Première traduction en code

```
/* tab-conversions1.c, imprime la table de
conversion des euros en devises,
version avec valeurs en dur */

#include <stdio.h>

main(){
   float euro, devise;

   euro=0;
   while (euro <= 200) {
        devise = euro * 2.11;
        printf("%4.0f %6.1f\n", euro, devise);
        euro = euro + 10;
   }
}
```

## Une version avec des constantes

... mais, c'est très mauvais de projeter dans un programme des nombres comme 0, 200, 2.11 etc sans préciser ce qu'ils représentent. Il vaut mieux définir des constantes :

```
#define NomConstante ChaîneDeCharactères
```

Le compilateur remplacera chaque occurrence du nom de la constante par la chaîne correspondante (*preprocessing*).

```
/* tab-conversions2.c, imprime la table de
conversion des euros en devises,
version paramétrable avec constantes */

#include <stdio.h>

#define DEBUT 0
        /* limite inférieure de la table */
#define FIN 200
        /* limite supérieure de la table */
#define STEP 10
        /* la taille du pas */
#define COURS 2.11

main(){
   float euro, devise;

   euro=DEBUT;
   while (euro <= FIN) {
        devise = euro * COURS;
        printf("%4.0f %6.1f\n", euro, devise);
        euro = euro + STEP;
   }
}
```

Encore une version, avec une autre sorte de boucle (`for`) :

```
/* tab-conversions3.c, imprime la table de
conversion des euros en devises, version avec
constantes et boucle for */

#include <stdio.h>

#define DEBUT 0
        /* limite inférieure de la table */
#define FIN 200
        /* limite supérieure de la table */
#define STEP 10
        /* la taille du pas */
#define COURS 2.11

main(){
  float euro;

  for (euro=DEBUT; euro <= FIN; euro = euro + STEP)
       printf("%4.0f %6.1f\n", euro, euro * COURS);
}
```

Dernière version, avec opérateurs spécifiques d'affectation :

```
/* tab-conversions4.c, imprime la table de
conversion des euros en devises,

version avec constantes et boucle for */


#include <stdio.h>

#define DEBUT 0
        /* limite inférieure de la table */
#define FIN 200
        /* limite supérieure de la table */
#define STEP 10
        /* la taille du pas */
#define COURS 2.11


main(){
   float euro;

   for (euro=DEBUT; euro <= FIN; euro += STEP)
                    /* euro = euro + STEP */
       printf("%4.0f %6.1f\n", euro, euro * COURS);
}
```

D'autres opérateurs se rapportant aux affectations sont possibles, par exemple :

`a *= 5;`

`b -= a + 4;` /* qui équivaut à `b = b-(a+4);` */

## Structure d'un programme complexe

Les constituants d'un programme sont :

- Les fichiers. Un programme peut être constitué de plusieurs fichiers, *liés* à la compilation. L'un d'entre eux contient le `main()`.
- Les fonctions. Une fonction est définie dans un fichier (qui peut contenir une ou plusieurs fonctions). Elle est constituée :
  - d'une en-tête qui définit les arguments et la valeur (le type de valeur) retournée par la fonction
  - du corps de la fonction, représenté par un bloc.
- Les blocs. Un bloc est une suite d'instructions à exécuter séquentiellement. Il est entouré par des accolades `{ ... }`.

## Les blocs

Un bloc a la forme suivante :

```
{
  déclarations de variables

  utilisables dans tout le bloc.

  instructions

  qui correspondent à l'exécution du bloc et exploitent les variables.
}
```

- un bloc peut contenir d'autres blocs.
- un bloc sans définition de variables, et à une seule instruction n'a pas besoin d'accolades.

## Exemple de bloc

```
{ /* commentaire : debut du bloc */

   int a = 4;    /* definitions */
   int b, c;

   b = a * 2;    /* une instruction */
   if (b > 7)    /* UNE instruction */
   {             /* conditionnelle  */
      int d;
      d = a + 3;
      c = 9 * d;
   }
   else
      c = 1;    /* ... qui finit ici */
} /* fin du bloc */
```

## Exécution d'un bloc

L'exécution d'un bloc correspond à un *contexte d'exécution :*

- Quelles sont les variables définies lorsque le bloc s'exécute ? (les variables définies au début du bloc)
- Les instructions sont exécutées séquentiellement, en partant de la première après les définitions.
- A la fin de l'exécution du bloc, son contexte est effacé : *les variables n'existent plus !!!* --> elles ne sont plus visibles de l'extérieur, et la place mémoire qu'elles occupaient est libérée.

Corollaire : un nom de variable correspond à une zone mémoire (une adresse) définie dans un contexte d'exécution.

*Deux variables de même nom définies dans 2 blocs différents n'ont rien à voir entre elles !!!*

## Fonctions

En gros : *une fonction équivaut à un bloc,*
mais c'est plus que cela : une fonction a un nom --> elle peut être *appelée* (exécutée) plusieurs fois dans un même programme.

```
void f1()      void f2()              main()
{               {                     {
   int a;          int b;                f2();
                                         f1();
   a = 3;          b = 8;                f2();
}               }                     }
```

Une fonction peut être appelée de n'importe quelle autre fonction (y compris d'elle-même). Elle est alors *évaluée*.

(exécution d'un programme C = évaluation de `main()`)

## Valeur de retour

Une fonction est un bloc nommé *qui peut retourner une valeur.*

La valeur retournée est *typée*.

L'instruction qui permet de retourner une valeur est :

```
return expression ;
```

(`expression` doit être du type prévu !!!)

```
int f1()                    void f2()
{                           {
   int a;                      int c = 4, toto;
   a = 7;                      toto = f1()+c*f1();
   if (a < 12)              }
      return 4;
   a += 3;
   return a + 8;
}
```

```
float convert()
{
   float montant;
   float cours;
   montant=200;
   cours=2.11;

   return montant * cours;
}
```

Remarque : dans `f1()`, `a` vaut toujours 7 au départ ..., dans `convert()`, il serait utile de pouvoir changer de montant et de cours...

## Paramètres

Une fonction peut avoir des paramètres. Le résultat de son exécution dépend alors de la valeur de ses paramètres.

```
#include <stdio.h>

float convert(float montant, float cours)
{
      return montant * cours;
}

main()
{
   float euro, cours_dollar, cours_livre;

   euro=100.0;
   cours_dollar=1.1;
   cours_livre=1.6;
   printf("%6.1f euro vaut %6.1f dollars\n", euro,
                        convert(euro,cours_dollar));
   printf("%6.1f euro vaut %6.1f livres\n", euro,
                        convert(euro,cours_livre));
}
```

## Syntaxe

On peut passer la valeur de plusieurs variables à une fonction :

```
void plusieurs(arg1, arg2, arg3, arg4)
int arg1;
char arg2;
int arg3, arg4;
{ .... }

main()
{
   int truc = 18;

   plusieurs(12, 'a', 2 - 89, truc);
}
```

... avec des types différents.

Les arguments arg1, ..., arg4 dans la déclaration de la fonction sont appelés les *paramètres* formels, alors que ceux avec lesquels la fonction est appelée sont ses *paramètres* effectifs.

Attention, différences de notation des en-têtes en K&R ou C ANSI

| | |
|---|---|
| `void KR(a, b)`<br>`int a;`<br>`char b;`<br>`{…}` | `void C_ANSI(int a, char b)`<br>`{…}` |

## Paramètres de fonctions : passage par valeur

Les arguments des fonctions sont toujours passés *par valeur* en C : l'exécution du bloc de la fonction est faite sur des *copies* locales des variables (venant du bloc appelant la fonction). Toute modification d'un argument (c'est-à-dire de sa copie) dans le bloc de la fonction n'a aucun effet sur l'original après le retour de la fonction.

### Exemple

```
void inc(int i)
{
  i = i + 1;
}

main()
{
  int x = 5;
  inc(x);

  /* x vaut toujours 5 !!! */
}
```

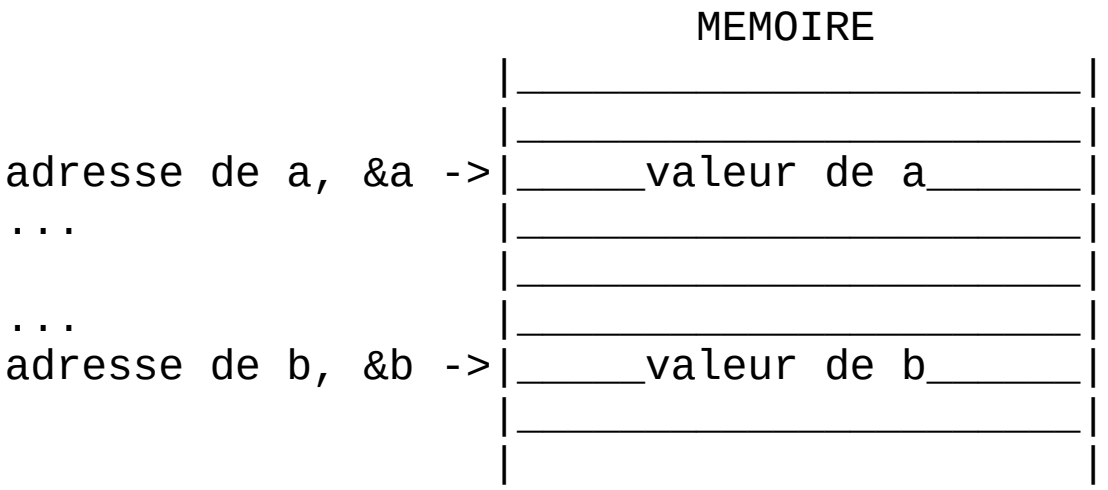
La variable x dans l'appel `inc(x)` sert à initialiser la valeur du paramètre formel de la fonction `inc`.

La fonction `inc` travaille sur une variable locale `i` !

## Paramètres de fonctions : suite

Mais comment faire si on a besoin de modifier des arguments d'une fonction ?

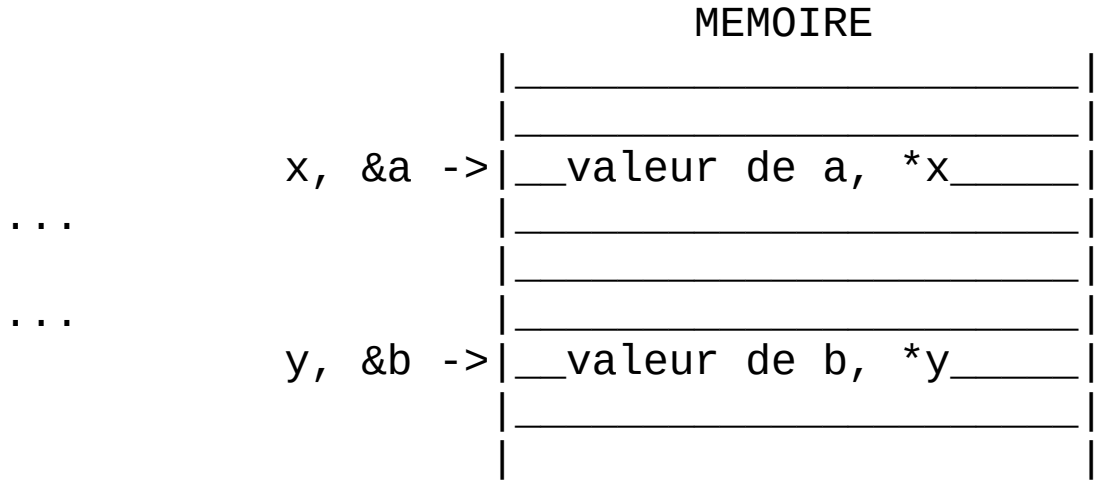
Par exemple, un programme de tri peut échanger les valeurs de deux éléments qui ne sont pas rangés dans le bon ordre, à l'aide de la fonction nommée `swap`. A cause du passage des paramètres par valeur, l'appel `swap(a,b);` n'est pas suffisant car `swap` ne peut modifier les arguments `a` et `b` dans le programme appelant. Le moyen de s'en sortir consiste à passer à la fonction `swap` les pointeurs (adresses mémoire, notées `&a` et `&b`) qui indiquent où sont rangées les valeurs de `a` et de `b`.




```
main()     /* programme appelant swap */
{
   int a,b;
    ...
   swap(&a,&b);
               /* les adresses de a et de b
                  sont passées à swap */
     /* les contenus de a et b sont échangés */
   ...
}
```

La fonction `swap` aura donc deux paramètres formels `x` et `y` qui sont deux adresses mémoire et échangera leurs contenus, c'est-à-dire, `*x` et `*y`.
La syntaxe est la suivante :

```
void swap(int * x, int * y)
     /* x et y sont des pointeurs (adresses),
       swap échange les valeurs pointées par x
       et y, c'est-à-dire *x et *y  */
{
    int temp;
    temp = *x;

    *x = *y;
    *y = temp;
}
```



- l'opérateur d'indirection * signifie «ce qui est pointé par» ;
- l'opérateur d'adresse & signifie «adresse de» ;

```
int i, j, * pi, * pj;
     /* pi et pj sont déclarés
       comme pointeurs sur des entiers,
       cela veut dire que si on applique
       l'opérateur * sur pi ou sur pj,
       on obtiendra une valeur du type int   */

  pi = &i;       /* pi pointe sur i, pi a pour
                  valeur l'adresse de i */
  j = *pi + 1;  /* équivaut à j = i + 1;  */

  *pi = 0;       /* équivaut à i = 0;  */

  pj = pi;       /* veut dire que pj pointe
                     aussi sur i  */
```

Attention : dans une déclaration de pointeur comme

```
int * pi;
```

l'étoile * n'est pas l'opérateur d'indirection.

## Un autre exemple : scanf

```
{
    int i;
    float x;

    scanf("%d %f",&i,&x);
    printf("%3d %6.1f",i,x);
}
```

avec en entrée la ligne

```
23456 12.123
```

`scanf` affectera `23456` à `i` et `12.123` à `x` et `printf` imprimera

```
234    12.1
```

## Instructions

Les instructions terminent toujours par des points-virgules `;`

## Instructions simples

- instruction vide : `;`
- expression : `a*5;  a<=1 && b>0;`
- appel de fonction : `convert(200,2.11);`

  `atoi("la chaine");`
- affectation `variable = expression` :

  `a = 4*6 + sqrt(8.7);`

  `a += 5;`

## Différentes catégories d'expressions, et d'opérateurs associés

- expressions arithmétiques (sur les nombres)

```
a*5     2.11/4.6+b*44
```

  - `+`, `-`, `*`, `/`, `%` (modulo), `<`, `<=`, `==` (égalité), `!=` (inégalité) , `>=`, `>`
  - les opérations peuvent faire intervenir différents types
  - les calculs se font avec une précision limitée

- expressions booléennes ou conditions

```
a == 3 && b < a    a <= 7 || a > 10
```

  - && ("et" logique), `||` ("ou" logique), `!` ("non"), `==` (équivaut)
  - utilisation des tables de vérité pour évaluer les expressions booléennes
  - convention : une expression (quelconque) est fausse si elle s'évalue à 0, sinon elle est interprétée comme vraie

- caractères `'a'` `'A'` `'Z'` (littéral entre quotes)
  - code ASCII du caractère, …
  - `<`, `<=`, `==`, `!=`, `>=`, `>`

## Instructions conditionnelles (if...else)

Pour exécuter des instructions si une condition est vérifiée:

```
if ( expression )
  bloc1
[
else
   bloc2
]
```

- `else` est optionnel
- `expression` est une expression booléenne (interprétée comme telle)

Rappel : un bloc est une instruction simple ou `{ ... }`

Exemples

```
main()
{
   int a = 8, b = 5;
        /* expr */
   if ( a+b > 16 ) { /* bloc 1 */
     a = a - 5;
     b = b + 4;
   }              /* fin bloc 1 */
   else a = 1;   /* bloc 2 -> a = 1; */
   a = a * 7;
   printf("%d %d\n",a,b);
}
```

Qu'affiche-t-on ?

Attention : si plusieurs `if`'s imbriqués, chaque `else` est associé au `if` qui le précède.

```
/* attention piège */

int moins(int a,b)
{
   if ( a-b <= 0 )
      a = -1;
      if ( a - b == 0 )
         a = 0;
   else
      a = 1;
   return a;
}
```

```
/* version corrigée */

int moins(int a,b)
{
   if ( a-b <= 0 ) {
      a = -1;
      if ( a - b == 0 )
        a = 0;
    }
   else
       a = 1;
   return a;
}
```

Combien valent dans les deux cas :

`moins(2,5) ?`

`moins(2,2) ?`

`moins(5,2) ?`

# Instructions itératives (`while` et `do...while`)

Elles servent à répéter plusieurs fois un bloc. La syntaxe est la suivante :

```
/* tant que expression faire bloc */

while ( expression )
   bloc
```

```
/* faire bloc tant que expression */

do
   bloc
while ( expression ) ;
```

- `expression` Booléenne
- Intérêt du `do` … `while` : pas d'initialisation des variables utilisées dans la condition d'arrêt.

Exemples (`while`)

```
/* n! (factorielle) */

 int n = res = 10;

while (n > 1) {
    n-=1;
    res *= n;
  }
```

```
/* ATTENTION ! (expression invariante) */

int i = 1, j = 3, t = 4;

while (i != 0) {
    t += 4;
    j -= 1;
}
```

Exemples (`do ... while`)

```
{
  int a = 0, b;

 do
    scanf("%d", &b);
    a += b;
    printf("a vaut maintenant %d\n",a);
 while (b > 0);
}
```

```
{
 char c;
 do
    c = getchar();
 while (c != EOF);
}
```

- rappel : `scanf` permet de lire un nombre au clavier ;
- `getchar()` renvoie un caractère lu au clavier ;
- EOF (pour End Of File) est une constante qui indique la fin du fichier.

## Instructions itératives (`for`)

Pour répéter plusieurs fois un bloc *en utilisant un (des) compteur(s)*.

```
for (expr initiale; expr booléenne; expr fin de bloc)
   bloc
```

- `expr initiale` est évaluée *une seule fois*, avant l'exécution du bloc `bloc`.
- `expr booléenne` est évaluée *avant chaque* exécution du bloc `bloc`. Si elle est fausse, la boucle est terminée et on n'exécute plus le bloc.
- `expr fin de bloc` est évaluée *après chaque* exécution du bloc `bloc`.

Généralement, `expr initiale` et `expr fin de bloc` sont des affectations d'une variable appelée *compteur*.

## Exemples (`for`)

### Produit des n premiers entiers

```
cumul = 1;

for (i=n; i>0; i-=1)
  cumul *= i;
```

ou :

```
for (cumul=1, i=n; i>0; i--)
  cumul*=i;
```

ou :

```
i = cumul = 1;

for ( ; i<=n; )
  cumul *= i++;
```

### Boucles infinies (2 versions)

```
for ( ; ; )
   ;
```

```
while (1)
  ;
```

### Exercice : Que vaut c après la boucle ?

```
for (i=1, j=1, c=0; i>=j; i++, j*=2)
  c += i - j;
```

## Instructions itératives (sortie anticipée : `break`)

Il est possible d'interrompre l'exécution d'une boucle ( `while`, `do` ... `while` et `for`) avant la fin en plaçant l'instruction break.

### Exemples

```
int cumul=0, i, j;

for (i=0; i<10; i++) {
  scanf("%d", &j);
  if (j<0)
    break;
  cumul += j;
}
```

## Instructions itératives (rebouclage anticipé : `continue`)

Il est possible de ne pas finir l'exécution du bloc de la boucle ( `while`, `do` ... `while` et `for`) si une condition est vérifiée en plaçant l'instruction `continue`.

### Exemples

```
int cumul=0, i, j;

for (i=0; i<20; i++) {
  scanf("%d", &j);
  if (j==7 || j==8)
    continue;
  cumul += j;
}
```

```
int cumul=0, i, j;

for (i=0; i<20; i++) {
  scanf("%d", &j);
  if ( (j%3) == 0 )
       /* modulo 3 */
    continue;
  cumul += j;
}
```

Dans le cas du `for`, le rebouclage est précédé de la ré-évaluation de la condition d'arrêt.

## Choix multiple (instruction `switch`)

Un `swich` permet d'effectuer différentes instructions suivant la valeur d'une expression

```
switch (expression) {
case etiquette1 : instructions
case etiquette2 : instructions
 ...
default: instructions  /* optionnel */
}
```

Les différents choix sont étiquetés par des expressions constantes qui doivent être toutes différentes.

La valeur de `expression` est comparée successivement aux étiquettes. Pour tous les cas où elle correspond à une étiquette, les instructions correspondantes sont exécutées.

Si aucun cas ne correspond, les instructions qui suivent le cas `default` (s'il existe) sont exécutées, sinon il ne se passe rien.

### Exemples

```
int n, a=8;

...
switch (n) {
  case 1: a++;
  case 2: a--;
  case 3: a*=2;
}
```

Quels sont les différents cas ? Que vaut `a` après le `switch` ?

```
char c; int chiffreHexa;
…
switch (c) {
  case '0': case '1': case '2': case '3':
  case '4': case '5': case '6': case '7':
  case '8': case '9':
    chiffreHexa = c - '0';
    break;
  case 'A': case 'B': case 'C':
  case 'D': case 'E': case 'F':
    chiffreHexa = c - 'A' + 10;
    break;
default: printf("Erreur !");
}
```

## Branchement inconditionnel (`goto`)

A ne PAS utiliser !

Permet d'aller à une étiquette (instruction étiquetée).

```
  goto etiquette;     /* avant l'étiquette */

…
  etiquette : instructions
…
  goto etiquette;     /* après l'étiquette */
…
```

Exemple :

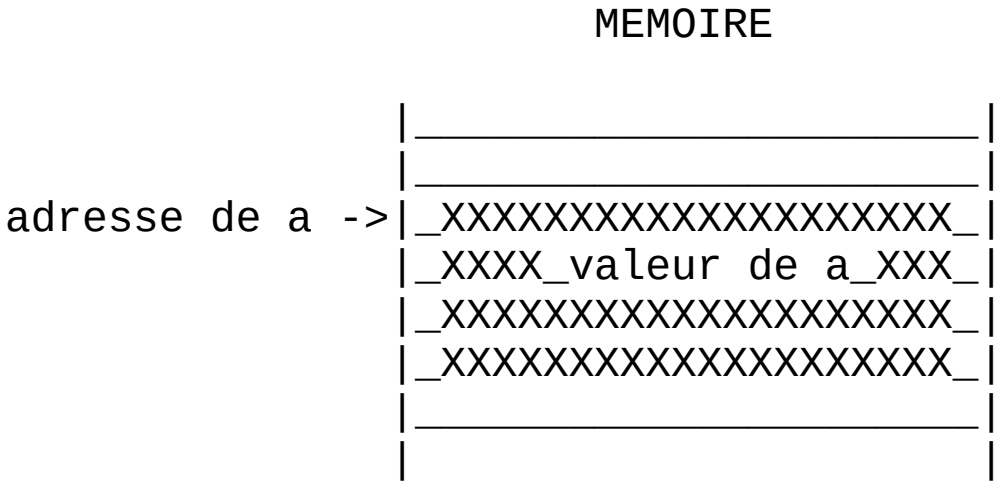
```
for (...)
   for (...) {
       ...
       if (catastrophe)
            goto erreur;       /* saut */
   }
...

erreur : printf("gros problème!!!");
                              /* étiquette */

...
```

# Types de données et gestion de la mémoire

Une déclaration de variable permet au compilateur de réserver (allouer) la place nécessaire au contenu de la variable. La place dépend du type de la variable.

L'opérateur `sizeof( unType )` permet de connaître le nombre d'octets prévu pour une variable de type `unType`.

L'endroit où est situé le premier octet alloué à une variable est l'*adresse* de la variable.




La définition `int a;` réserve 4 octets pour stocker la valeur de `a` (`sizeof(int)` vaut 4).

## Pointeurs (rappel)

Les pointeurs sont des variables ayant comme valeurs des adresses.

Ils pointent vers des variables d'un type donné, indiqué à la définition du pointeur `type * variable; .`

```
int * ptr1;

char * ptr2;

int * *ptr3;
```

Soit une variable `variable`, de type `type` :

- L'expression `&variable` renvoie l'adresse de la variable, qui est donc de type `type *`, c'est à dire *pointeur de* `type`.

- L'expression `*variable` renvoie la valeur de l'adresse que contient `variable`. `variable` est donc nécessairement un pointeur. L'expression est du type pointé par `type`.

Exemples

```
int a;

int * ptr;    /* déclaration de ptr qui est une
                 variable qui contient
                 l'adresse d'un entier */
a = 10;

ptr = & a;   /* ptr vaut l'adresse (&) de a */

*ptr = a + 2;
```

Que vaut `a` ? Que vaut `*ptr` ?

```
int i = 249;

int *ptr = &i;
```

Que valent les expressions :

```
1) i + 5;          5) *ptr + 5;
2) *(&i);          6) &(*i);
3) &ptr;           7) *(ptr + 5);
4) **ptr;          8) i**ptr;
```

De quel type sont ces expressions ?

## Place Mémoire

Comment est réservée la mémoire ?

- Si la taille est connue à la compilation, une définition de variable suffit ;
- Sinon, réservation explicite via des pointeurs.

Objets complexes :

- Tableaux : ensemble d'objets identiques ;
- Structures : assemblage d'objets de plusieurs types différents.

## Tableaux à une dimension (vecteurs)

La syntaxe de la déclaration est la suivante :

```
type nomVariable[taille];
```

```
float b[20];

char toto[N+1];

int *tutu[5];
```

La place mémoire allouée à la variable `nomVariable` est de `taille * sizeof(type)`.

L'accès à un élément : `nomVariable[index]`.L'index est une expression entière.

Attention : Les index débutent à 0.

Exemples

```
{
  int t[6];
  int i;

  for (i=0; i<6; i++) t[i] = 0;
  for (i=0; i<5; i++) t[i] = t[i]+t[i+1]+i;
}
```

Après la déclaration, il n'y a plus aucun contrôle sur la taille du tableau alloué : *c'est au programmeur de savoir si le $i^{ème}$ élément de son tableau existe*.

Initialisation :

```
{
  int a[3] = { 1, 2, 3 };

  int a[3] = { 1, 2 /* , 0 */ };

  int a[] = { 1, 2, 3 };
}
```

Cas des chaînes de caractères. Syntaxe : `"Une Chaine"` .

```
{
  char chaine[] = "Une chaine";

  chaine[4] = 'C';
}
```

Une chaîne est un tableau de caractères plus le caractère `NULL` (code ASCII )

| U | n | e |  | C | h | a | i | n | e | \0 |
|---|---|---|---|---|---|---|---|---|---|---|

## Tableaux multidimensionnels

Déclaration :

```
{
  int a[3][6];
  char tableau[15][9][56];
  int a[2][3] = { { 1, 2, 3 }, { 4, 5, 6}};
}
```

Accès :

```
{
  int i, j;
  int N=5, M=8;

  int tableau[N][M];

  /* initialisation */
  ...

  /* affichage */

  for (i=0; i<N; i++) {
    for (j=0; j<M; j++)
      printf("%d ", tableau[i][j]);
    printf("\n");
  }
```

## Tableaux et pointeurs

Si `tab[10]` est un tableau, `tab` est en fait un pointeur constant (une adresse fixe en mémoire), dont la valeur est `&tab[0]`, l'adresse du premier élément du tableau.

```
int tab[10];
int *ptr = &tab[0]; /* == tab */
```

D'où :

`tab[0] == *tab`

`tab[1] == *(tab + 1)`

`tab[i] == *(tab + i) == tab[i]`

Décalage de `i * sizeof(type des éléments)` pour calculer l'adresse de la valeur `tab[i]`.

## Structures

Une *structure* regroupe un ensemble d'objets de types différents ou identiques. Chaque structure est associée à un type.

Il faut d'abord définir le type associé à la structure pour pouvoir ensuite définir des variables de ce nouveau type.

```
struct nomType{
 type1 nomChamp1;
 type2 nomChamp2;
 ...
};
```

```
struct point{
 int x;
 int y;
 double norme;
};
```

```
struct personne{
   char * nom;
   int age;
};
```

Déclaration de variables :

```
struct nomStructure variable;
```

Exemples :

```
/* les definitions de types sont faites en DEHORS des
blocs */

struct point {
  int x, y;
  double norme;
};

struct personne {
   char * nom;
   int age;
};

main()
{
  int i, j;
  struct point unPoint;
  struct personne l, m;
  struct point p1 = { 3, 4, 5.0 };
  struct personne toto = { "Mr Toto", 10 };
  ...
}
```

La notation `.` (point) est utilisée : pour accéder à la valeur du champ `age` d'une variable `toto` de type `struct personne` :

```
{
  int sonAge;
  char *sonNom;
  struct personne toto = { "Mr Toto", 10 };
  sonAge = toto.age;       toto.age = 24;

  sonNom = toto.nom;

}
```

La notation `->` (flèche) est utilisée si la variable est un pointeur de structure :

```
{
 int sonAge;
 struct personne toto = {"Mr Toto",10};
 struct personne *pointeurToto = &toto;
 sonAge = pointeurToto->age; /* toto.age */

 /* ou (*pointeurToto).age */

}
```

## Allocation dynamique de mémoire

Réservation implicite :

```
int i;  /* réserve la place pour un entier, statique */

int tab[10] /* réserve la place pour 10 entiers, statique */
```

Comment réserver de la place mémoire de manière explicite ? Par exemple :

*Combien de notes ? (lecture de nbNotes)*

*Pour i de 1 à nbNotes*

*notes[i]* ß *lecture d'une note*

*moyenne(notes, nbNotes)*

Problème : comment garantir que le tableau *notes* est assez grand ?

1. Allouer statiquement de la place mémoire :

   ```
   int notes[MAX] ;
   ```

   - si plus de MAX entiers ?
   - si nbNotes << MAX, mémoire gachée ...

2. Allouer dynamiquement de la place en mémoire une fois que l'on connaît la taille nécessaire.

## Réservation de la mémoire ( fonction `malloc()` )

L'allocation dynamique de mémoire utilise les pointeurs. Principe :

- Le programme demande de la place en mémoire (en octets),
- Le système recherche une zone mémoire libre assez grande, et renvoie l'adresse du premier octet de la zone, c'est-à-dire, un pointeur sur la zone allouée.

Utilisation de la fonction `malloc()` : `void * malloc(int)`

- la fonction a un argument de type entier (nb octets demandés),
- elle renvoie un pointeur de caractères (adresse zone allouée).

Exemple

```
int *ptr1;

ptr1 = malloc(10 * sizeof(int));

ptr1 = (int *) malloc(10 * sizeof(int));
                      /* avec cast OK  */

ptr1[9] = 1234;
```

Exemple plus grand

```
main(){
 int nom, nbNoms = 0;
 char *tabMinus[] = { "toto", "mimi", "lulu", 0 };
 char **tabMajus;

 while (tabMinus[nbNoms] != 0)
   nbNoms++;
 tabMajus = (char **)malloc(nbNoms*sizeof(char*));
 for (nom=0; nom<nbNoms; nom++) {
   int car;
   int taille = strlen(tabMinus[nom]);
   tabMajus[nom] = (char *)malloc(taille+1);
   for (car=0; car<taille; car++)
     tabMajus[nom][car] = tabMinus[nom][car];
   *tabMajus[nom] = *tabMinus[nom] +'A' - 'a';
   tabMajus[nom][taille] = 0;
 }
 for (nom=0; nom<nbNoms; nom++)
   printf("%s\n",tabMajus[nom]);

}
```

Que fait le programme ?

## Libération de la mémoire ( fonction `free()` )

L'allocation de la mémoire est explicite et la libération l'est aussi.

La fonction `free(char *)` déalloue la mémoire allouée avec `malloc()`.

allocation implicite (statique) :

```
{
  int tab[8];
  tab[3] = 4;
  ...

  /* liberation automatique */
}
```

allocation explicite (dynamique) :

```
{
  int *tab=(int *)malloc(8*sizeof(int));
  tab[3] = 4;
  ...
  free(tab);
}
```

## Fonctions et pointeurs

Une fonction et son appel :

```
double carre(double x)
{
  int res;

  res = x * x;
  return res;
}

main()
{
  double res = 56;
  double truc;

  truc = carre(res);
  res = res * truc;
}
```

Déroulez le programme ...

Comment écrire `carreEtCube(double x)` qui renvoie à la fois le carré et le cube de `x` ?

Première solution

Une fonction C renvoie au plus une valeur : on peut renvoyer une structure contenant les deux valeurs $x^2$ et $x^3$ :

```
struct cEtc { /* definition du type */
  double carre;
  double cube;
};

/* fonction */
struct cEtc carreEtcube(double x)
{
  struct cEtc res;

  res.carre = x * x;
  res.cube = res.carre * x;
  return res;
}

main()
{
  double res = 56;
  struct cEtc truc;

  truc = carreEtcube(res);
}
```

Il faut définir un type spécialement ...

Deuxième solution

Faire du passage par adresse : la fonction renvoie $x^3$ et stocke $x^2$ dans un argument :

```
double carreEtcube(double x, double carre)
        /*  FAUX !!! */
{
  carre = x * x;
  return carre * x;
}

main()
{
  double truc, res;

  truc = carreEtcube(56, res)
}
```

Ça ne marche pas : `res` ne contient pas la bonne valeur après l'appel dans le `main()` ...

```
double carreEtcube(double x, double * carre)
{
  *carre = x * x;

  return (*carre) * x;
}

main()
{
  double truc, res;

  truc = carreEtcube(56, &res)
}
```