



Université Chouaïb Doukkali
Faculté des Sciences El Jadida



PROGRAMMATION EN LANGAGE C

SMI – S4
FS – EL JADIDA

Pr. BELAQZIZ Salwa
Salwa.belaqziz@gmail.com

2017-2018

Programmation en langage C

Plan

1

Rappels (Types de bases, variables, opérateurs, structures de contrôles,..)

2

Les fonctions

3

Les pointeurs et l'allocation dynamique

4

Les chaines de caractères

5

Les types composés (structures, unions, synonymes)

6

Les fichiers

Programmation en langage C

Plan

1

Rappels (Types de bases, variables, opérateurs, structures de contrôles,..)

2

Les fonctions

3

Les pointeurs et l'allocation dynamique

4

Les chaines de caractères

5

Les types composés (structures, unions, synonymes)

6

Les fichiers

1

Rappels

Présentation du langage C



Ken Thompson (à gauche)
et Dennis Ritchie (à droite).

- Langage de programmation développé au cours de l'année 1972 par Dennis Ritchie et Ken Thompson.
- Un Langage polyvalent permettant le développement de systèmes d'exploitation, de programmes applicatifs scientifiques et de gestion.
- Un Langage structuré.
- Portabilité du code source, due à l'emploi de bibliothèques dans lesquelles sont reléguées les fonctionnalités liées à la machine.
- Grande efficacité et puissance.

1

Présentation du langage C

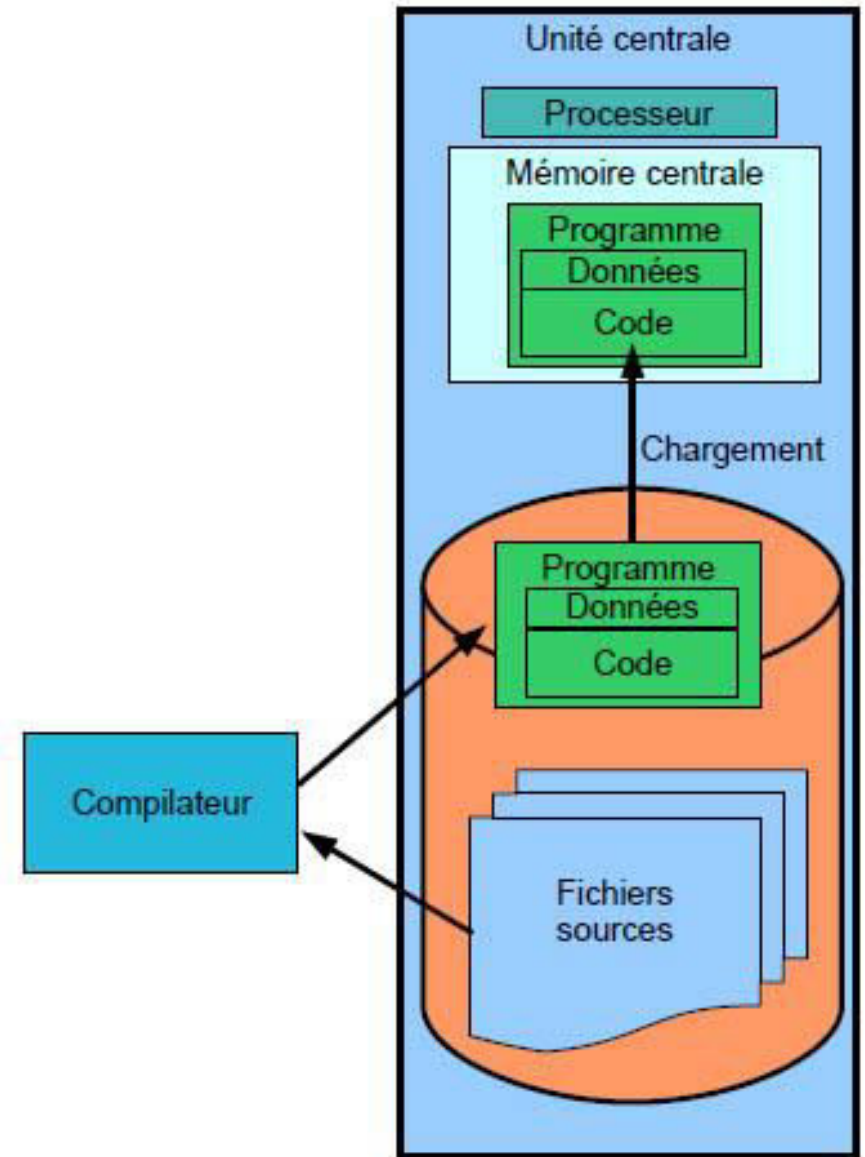
Rappels

Pour pouvoir programmer en C, vous devez nécessairement avoir installé un compilateur C sur votre machine.

Après avoir été compilé, le programme peut être exécuté en chargeant sa version exécutable en mémoire

Le compilateur est un programme exécutable qui permet de traduire un programme écrit dans un langage de programmation en langage machine.

Compilation et exécution



1

Présentation du langage C

Rappels

Utilisation des **EDI (Environnement de Développement Intégré)** pour le développement des programmes :

- peut être adapté à un langage de programmation particulier ou à plusieurs langages
- comprend en général un **éditeur de texte** spécialement adapté au langage, un **compilateur** un **débogueur** (outil de mise au point) et un outil de développement d'interface graphique.

A screenshot of the Dev-C++ 4.9.9.2 IDE. The main window shows a C program named 'hello.c' with the following code:

```
#include <stdio.h>

int main() {
    printf("Hello, World!\n");

    system("pause");
    return 0;
}
```

The bottom window shows the execution of the program, displaying 'Hello, World!' and 'Appuyez sur une touche pour continuer...'. The title bar of the bottom window reads 'C:\Documents and Settings\philippe\Mes documents\c_ers2\hello.exe'.

1

Rappels

Structure d'un programme C

Programme C : composé d'une ou plusieurs fonctions dont l'une doit s'appeler **main**

Type de la valeur de retour → **int** **main**(**void**)

Nom de la fonction (main signifie principale, ses instructions sont exécutées) →

Liste des arguments entre parenthèses →

Début → {

/* corps du programme */

déclaration des constantes et des variables ;

instruction1 ;

instruction2 ;

...

Fin → }

Entre accolades "{" et "}" on met la succession d'instructions à réaliser.(Bloc)

1

Rappels

Structure d'un programme C

Programme C : composé d'une ou plusieurs fonctions dont l'une doit s'appeler **main**

void main(void) : La fonction main ne prend aucun paramètre et ne retourne pas de valeur.

int main(void) : La fonction main retourne une valeur entière à l'aide de l'instruction return (0 si pas d'erreur).

int main(int arg1, char arg2) : On obtient alors des programmes auxquels on peut adresser des arguments au moment où on lance le programme.

1

Rappels

Structure d'un programme C

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

```
#define PI 3.14
```

```
int fonc_somme(int a, int b);
```

```
int fonc_division(int a, int b);
```

```
void main()
```

```
{ /* début du bloc de la fonction main*/
```

```
    int i, j; /* définition des variables locales */
```

```
    i = 10 ;
```

```
    j=2;
```

```
    fonc_somme(I,j) ;
```

```
    fonc_division(i,j) ;
```

```
} /* fin du bloc de la fonction main */
```

```
int fonc_somme(int a, int b) {
```

```
    return (a+b);
```

```
}
```

```
int fonc_division(int a, intb) {
```

```
    return (a / b);
```

```
}
```

Directives du préprocesseur :
accès avant la compilation

Déclaration des fonctions

Programme principal

Définitions des
fonctions

1

Préprocesseur

Rappels

- Le préprocesseur effectue un prétraitement du programme source avant qu'il soit compilé.
- Ce préprocesseur exécute des instructions particulières appelées **directives**.
- Ces directives sont identifiées par le caractère **#** en tête.

Inclusion de fichiers

```
#include <nom-de-fichier>      /* répertoire standard */  
#include "nom-de-fichier"      /* répertoire courant */
```

```
La gestion des fichiers (stdio.h)           /* Entrees-sorties standard */  
Les fonctions mathématiques (math.h)  
Taille des type entiers (limits.h)  
Limites des type réels (float.h)  
Traitement de chaînes de caractères (string.h)  
Le traitement de caractères (ctype.h)  
Utilitaires généraux (stdlib.h)  
Date et heure (time.h)
```

1

1^{er} programme

Rappels

Cours de programmation en langage C
"SMI-S4"

Quel est ton âge ? 18

ton âge est de 18 ans

#include <stdio.h>

#include <conio.h>

int main(void)

{

int age; /*déclaration d'une variable*/

printf("Cours de programmation en \t langage C,\n \"SMI-S4\" \n");

printf("Quel est ton âge? ");

scanf(" %d", &age); /* lecture de l'âge, on donne l'adresse de age */

printf(" \n ton âge est de %d ans \n",age);

getch(); /* Attente d'une saisie clavier */

return 0; /* En principe un code d'erreur nul signifie "pas d'erreur". */

}

1

1^{er} programme

Rappels

Format des paramètres passés en lecture et écriture.

"%c" : lecture d'un caractère.

"%d" ou "%i" : entier signé.

"%e" : réel avec un exposant.

"%f" : réel sans exposant.

"%o" : le nombre est écrit en base 8.

"%s" : chaîne de caractère.

Les caractères précédés de \ sont interprétés comme suit :

*\\ : caractère *

\n : retour à la ligne

\t : tabulateur.

\\" : caractère "

L'utilisation de & est indispensable avec scanf (valeur lue et donc modifiée), pas avec printf (valeur écrite et donc non modifiée).

1

Les variables

Rappels

Déclarations

Syntaxe : Type identificateur1, identificateur2, ...,.... ;

Exemple: char c1, c2, c3;
 int i, j, var_ent;

Initialisations

Les variables doivent être déclarées avant leur utilisation dans un début de bloc (juste après {).

```
void main(void)
{
    char c;
    int i,j, k;
    c = 'A';
    i = 50;
    j =10;
    K=10;
```

est équivalent à

```
void main(void)
{
    char c = 'A';
    int i=50,j, k;
    j=k=10;
```

1

Rappels

Types de base sous le langage C

4 types de base, les autres types seront dérivés de ceux-ci.

| Type | Signification | Exemples de valeur | Codage en mémoire | Peut être |
|---------------|------------------------------|--|-------------------|-------------------------------|
| char | Caractère unique | 'a' 'A' 'z' 'Z' '\n' 'a' 'A' 'z' 'Z' '\n' Varie de -128 à 127 | 1 octet | signed, unsigned |
| int | Nombre entier | 0 1 -1 4589 32000 -231 à 231 +1 | 2 ou 4 octets | Short, long, signed, unsigned |
| float | Nombre réel simple | 0.0 1.0 3.14 5.32 -1.23 | 4 octets | |
| double | Nombre réel double précision | 0.0 1.0E-10 1.0 - 1.34567896 | 8 octets | long |

1

Rappels

Les opérateurs en langage C

Les opérateurs arithmétiques

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

Les opérateurs de comparaison

- | | |
|--------------|--------------------|
| < | plus petit |
| <= | plus petit ou égal |
| > | plus grand |
| >= | plus grand ou égal |
| == | égal |
| != | différent |

Les opérateurs logiques

- | | |
|-------------------|-------------------|
| && | et |
| | ou (non exclusif) |
| ! | non |

1

Incrément et décrétement

Rappels

- C a deux opérateurs spéciaux pour incrémenter (ajouter 1) et décrétement (retirer 1) des variables entières
 - ++** **increment**
 - **decrement**
- Ces opérateurs peuvent être *préfixés* (avant la variable) ou *postfixés* (après)

i++ est équivalent à i += 1 ou i = i + 1

1

Incrément et décrémentation

Rappels

Préfixe et Postfixe

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

```
int main(void)
```

```
{
```

```
    int    i, j = 5;
```

```
    i = ++j;
```

```
    printf("i=%d, j=%d\n", i, j);
```

```
    j = 5;
```

```
    i = j++;
```

```
    printf("i=%d, j=%d\n", i, j);
```

```
    return 0;
```

```
}
```

équivalent à:

1. j++;
2. i = j;

i=6, j=6

équivalent à:

1. i = j;
2. j++;

i=5, j=6

1

Rappels

Les structures de contrôle en C

Alternative:

`if-else`

Choix Multiple:

`switch-case`

Itérations:

`for, while, do-while`

Rupture de Contrôle: `break, continue, return ... goto`

1

Rappels

Les structures de contrôle en C

Les décisions : if - else

```
if (expression booléenne vraie)
{
    BLOC 1 D'INSTRUCTIONS
}
else
{
    BLOC 2 D'INSTRUCTIONS
}
```

Le bloc **"else"** est optionnel.

```
if (a<b)
{
    min=a;
}
else
{
    min=b;
}
```

1

Rappels

Les structures de contrôle en C

Les décisions : if - else

if (i == 10) i++; == et pas =

La variable i ne sera incrémentée que si elle est égale à 10.

if (!recu) printf ("rien reçu\n");

Le message "rien reçu" est affiché si reçu vaut zéro.

Si plusieurs instructions, il faut les mettre entre accolades :

```
if ((!recu) && (i < 10) && (n!=0) )
{
    i++;
    moy = som/n;
    printf(" la valeur de i =%d et moy=%f\n", i,moy) ;
}
else
{
    printf ("erreur \n");
    i = i +2;           // i +=2 ;
}
```

if(delta != 0) = if(delta)

if(delta == 0) = if(!delta)



1

Rappels

Les structures de contrôle en C

if emboîtés

else est associé avec le if le plus proche

```
int i = 100;
```

```
if(i > 0)
```

```
    if(i > 1000)
```

```
        printf("i > 1000\n");
```

```
    else
```

```
        printf("i inférieur à 1000\n");
```

i inférieur à 1000

1

Rappels

Les structures de contrôle en C

Les boucles : Les itérations – **for**

Syntaxe en C:

```
for( init ; test; increment)
{
    /* corps de for */
}
```

```
int i,j;

for (i = 0; i <3; i++) {
    printf ( "i = %d\n", i);
}
for(j = 5; j > 0; j- -)
    printf("j = %d\n", j);
```

```
i = 0
i = 1
i = 2
j = 5
j = 4
j = 3
j = 2
j = 1
```

1

Rappels

Les structures de contrôle en C

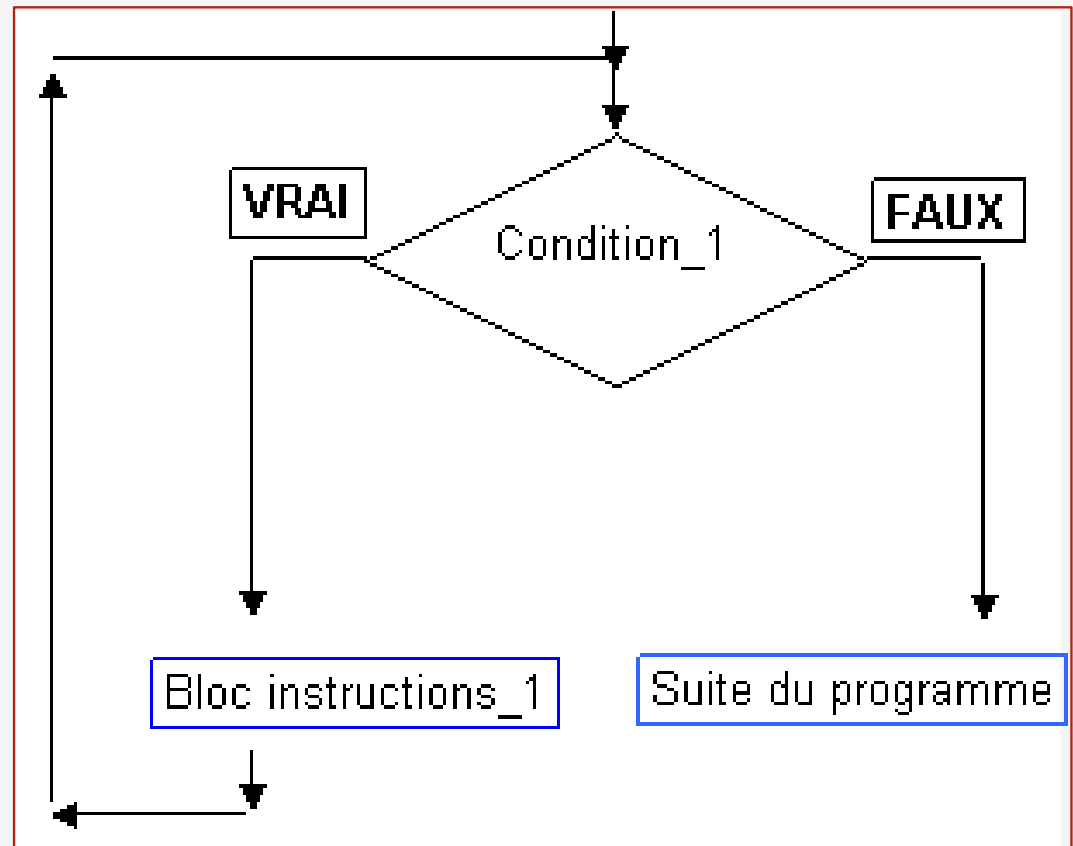
Les boucles : TANT QUE ... FAIRE... WHILE

Syntaxe en C:

```
while (expression)
{
.....;    /* bloc d'instructions */
.....;
.....;
}
```

Le test se fait **d'abord**, le bloc d'instructions n'est pas forcément exécuté.

Rq: les {} ne sont pas nécessaires lorsque le bloc ne comporte qu'une seule instruction.



1

Rappels

Les structures de contrôle en C

Les boucles : TANT QUE ... FAIRE... WHILE

Syntaxe en C:

```
while (expression)
{
    .....;    /* bloc d'instructions */
    .....;
    .....;
}
```

Le test se fait **d'abord**, le bloc d'instructions n'est pas forcément exécuté.

Rq: les {} ne sont pas nécessaires lorsque le bloc ne comporte qu'une seule instruction.

Exemple

```
i=1;
while(i<5)
{
    printf("Intérieur %d\n",i);
    i++;
}
printf("Extérieur %d\n",i);
```

| | |
|-----------|---|
| Intérieur | 1 |
| Intérieur | 2 |
| Intérieur | 3 |
| Intérieur | 4 |
| Extérieur | 5 |

1

Rappels

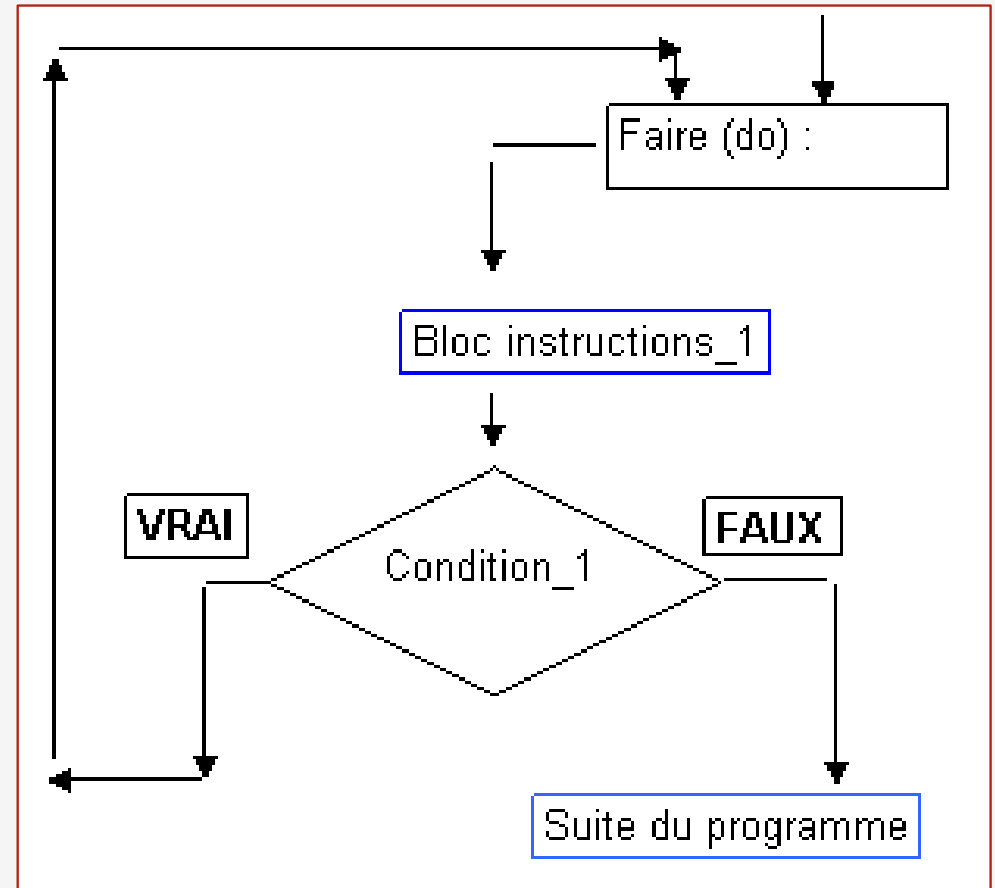
Les structures de contrôle en C

Les boucles : REPETER ... TANT QUE do while

Syntaxe en C:

```
do
{
    .....;          /* bloc
d'instructions */
    .....;
}
while (expression);
```

(garantit l'exécution au moins une fois)



1

Rappels

Les structures de contrôle en C

Les boucles : REPETER ... TANT QUE do while

Syntaxe en C:

```
do
{
    .....;          /* bloc
d'instructions */
    .....;
}
while (expression);
```

(garantit l'exécution au moins une fois)

Exemple

```
int j = 5;
do
    printf("j = %i\n", j--);
while(j > 0);
printf("stop\n");
```

j = 5
j = 4
j = 3
j = 2
j = 1
stop

1

Rappels

Les structures de contrôle en C

Choix multiples

AU CAS OU... FAIRE : **switch-case**

```
switch(variable de type char ou int)    /* au cas où la variable vaut: */
{
    case valeur1: .....;                /* cette valeur1(étiquette): exécuter ce bloc d'instructions.*/
        .....;
        break;                          /* L'instruction d'échappement break;
                                           permet de quitter la boucle ou l'aiguillage le plus proche.
                                           */

    case valeur2:.....;                  /* cette valeur2: exécuter ce bloc d'instructions.*/
        .....;
        break;

    .
    .
    .
    default: .....;                     /* aucune des valeurs précédentes: exécuter ce bloc
        .....;                          d'instructions, pas de "break" ici.*/
}
```

1

Rappels

Les structures de contrôle en C

Choix multiples

AU CAS OU... FAIRE : switch-case

```
main( )
{ char c;
  switch (c) {
    case 'b':
    case 'm':
    case 'g':
    case 'y': printf("voyelle\n");
              break ;
    default : printf("consonne\n");
  }
}
```

1

Rappels

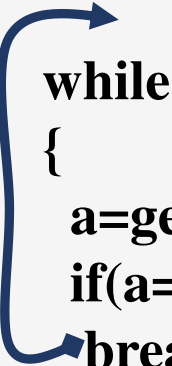
Les structures de contrôle en C

Instructions d'échappement

Pour rompre le déroulement séquentiel d'une suite d'instructions

BREAK: permet d'arrêter le déroulement de la boucle et le passage à l'instruction qui la suit

```
int i, j=1;
char a;
for (i = -10; i <= 10; i++){
    while(j!=0) /* boucle infinie */
    {
        a=getchar();
        if(a == 'x')
            break;
    }
}
```



En cas de boucles imbriquées, break ne met fin qu'à la boucle la plus interne

CONTINUE : permet l'abandon de l'itération courante et le passage à l'itération suivante

```
for (i = -10; i <= 10; i++)
{
    if (i == 0)
        continue;
    // pour éviter la division par zéro
    printf(" %f", 1 / i);
}
```

return (expression);

permet de sortir de la fonction qui la contient

1

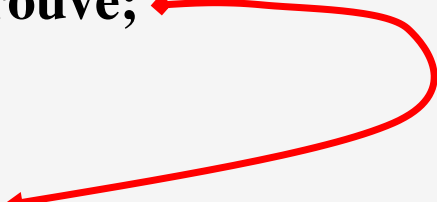
Rappels

Les structures de contrôle en C

Instructions d'échappement

goto étiquette

```
#include <stdio.h>
void main()
{
    int i, j;
    for (i=0; i < 10; i++)
        if ( i>4 )
            goto trouve;
    trouve:
    printf(" la valeur de i est %d\n",i);
}
```



1

Rappels

Les tableaux

Déclaration de tableaux

- Un tableau (**array**) est une **collection de variables de même type**, appelées éléments
- On les déclare par un type, un nom et une dimension (**CONSTANTE**) placée entre []
- Le C alloue toujours un tableau dans une zone contigüe de la mémoire
- Une fois déclaré, **on ne peut redimensionner un tableau**

Exemples

int tab[4]; déclare un tableau de 4 valeurs entières

| | | | |
|--------|--------|--------|--------|
| tab[0] | tab[1] | tab[2] | tab[3] |
|--------|--------|--------|--------|

float A[5] = { 10.1, 20.3, 30.5, 40.0, 50.4 };

1

Les tableaux

Rappels

Accès aux éléments d'un tableau

```
void main(void)
```

```
{
```

```
    int    a[6];
```

```
    int    i = 7;
```

```
    a[0] = 59;
```

```
    a[5] = -10;
```

```
    a[i/2] = 2;
```

```
    a[6] = 0;
```

```
    a[-1] = 5;
```

```
}
```

| a | |
|-----|---|
| 59 | 0 |
| ? | 1 |
| ? | 2 |
| 2 | 3 |
| ? | 4 |
| -10 | 5 |



Les tableaux consomment beaucoup de place mémoire. On a donc intérêt à les dimensionner au plus juste.

```
void main()
```

```
{
```

```
    const int N=10;
```

```
    int t[N],i;
```

```
    for (i=0;i<N;i++)
```

```
    {
```

```
        printf("Entrez t[%d]=",i);
```

```
        scanf("%d",&t[i]);
```

```
    }
```

```
}
```


1

Les tableaux

Rappels

Accès aux éléments d'un tableau

```
void main(void)
```

```
{
```

```
    int    a[6];
```

```
    int    i = 7;
```

```
    a[0] = 59;
```

```
    a[5] = -10;
```

```
    a[i/2] = 2;
```

```
    a[6] = 0;
```

```
    a[-1] = 5;
```

```
}
```

a

59

0

?

1

?

2

2

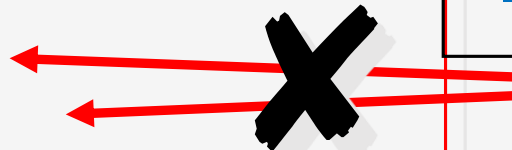
3

?

4

-10

5



```
void main(void)
```

```
{
```

```
    int i;
```

```
    int A[6] = { 1, 2,3, 5, 7, 11 };
```

```
    for (i=0;i<6;i++)
```

```
        printf(" %d ", A[i]);
```

```
}
```

Les tableaux consomment beaucoup de place mémoire. On a donc intérêt à les dimensionner au plus juste.

1

Les tableaux

Rappels

Exemple :

Calcul du nombre d'étudiants ayant une note supérieure à 10

```
main ( )
{ float notes[30];
  int nbre,i;
  for(i=0;i<30;i++)
    { printf ("Entrez notes[%d] \n ",i);
      scanf(" %f" , &notes[i]);
    }
  nbre=0;
  for (i=0; i<30; i++)
    if (notes[i]>10) nbre+=1;
  printf (" le nombre de notes > à 10 est égal à : %d", nbre);
}
```

Tableaux à plusieurs dimensions

On peut définir un tableau à n dimensions de la façon suivante:

Type **Nom_du_Tableau**[D1][D2]...[Dn];

où D_i est le nombre d'éléments dans la dimension i

Exemple : pour stocker les notes de 20 étudiants en 5 modules dans deux examens, on peut déclarer un tableau :

float **notes**[20][5][2];

(notes[i][j][k] est la note de l'examen k dans le module j pour l'étudiant i)

1

Rappels

Les tableaux

Tableaux à deux dimensions (Matrices)

Syntaxe : **Type** **nom_du_Tableau**[nombre_ligne][nombre_colonne];

Ex: **short A[2][3];** On peut représenter le tableau A de la manière suivante :

| | | |
|---------|---------|---------|
| A[0][0] | A[0][1] | A[0][2] |
| A[1][0] | A[1][1] | A[1][2] |

- Un tableau à deux dimensions $A[n][m]$ est à interpréter comme un tableau unidimensionnel de dimension n dont chaque composante tableau unidimensionnel de dimension m .
- Un tableau à deux dimensions $A[n][m]$ contient $n * m$ composantes. Ainsi lors de la déclaration, on lui réserve un espace mémoire dont la taille (en octets) est égal à : **$n * m * \text{taille du type}$**

1

Rappels

Les tableaux à deux dimensions

Initialisation à la déclaration d'une matrice

L'initialisation lors de la déclaration se fait en indiquant la liste des valeurs respectives entre accolades ligne par ligne

Exemple :

```
float A[3][4] = {{-1.5, 2.1, 3.4, 0}, {8, 7e-5, 1, 2.7 }, {3.1, 0, 2.5E4, -1.3E2}};
```

| | | | |
|----------------|---------------|----------------|----------------|
| A[0][0]=-1.5 , | A[0][1]=2.1, | A[0][2]=3.4, | A[0][3]=0 |
| A[1][0]=8 , | A[1][1]=7e-5, | A[1][2]=1, | A[1][3]=2.7 |
| A[2][0]=3.1 , | A[2][1]=0, | A[2][2]=2.5E4, | A[2][3]=-1.3E2 |

1

Rappels

Les tableaux à deux dimensions

Matrices : saisie et affichage

- Saisie des éléments d'une matrice d'entiers $A[n][m]$:

```
for(i=0;i<n;i++)
    for(j=0;j<m;j++)
        { printf ("Entrez la valeur de A[%d][%d] \n ",i,j);
          scanf(" %d" , &A[i][j]);
        }
```

- Affichage des éléments d'une matrice d'entiers $A[n][m]$:

```
for(i=0;i<n;i++)
    { for(j=0;j<m;j++)
      printf (" %d \t",A[i][j]);
    }
```

1

Rappels

Les tableaux à deux dimensions

Exercice

Ecrire un programme qui construit et affiche une matrice ***carrée unitaire*** U de dimension N (max 50).

Une matrice unitaire est une matrice, telle que:

**$U_{ij} = 1$ si $i=j$
sinon $U_{ij} = 0$**

1

Rappels

Les tableaux à deux dimensions

Exercice : correction

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

```
main()  
{
```

```
/* Déclarations */
```

```
int U[50][50]; /* matrice unitaire */
```

```
int N;      /* dimension de la matrice unitaire */
```

```
int I, J;   /* indices courants */
```

```
/* Saisie des données */
```

```
printf("Dimension de la matrice carrée (max.50) : ");
```

```
scanf("%d", &N);
```


1

Rappels

Les tableaux à deux dimensions

```
/* Construction de la matrice carrée unitaire */
```

```
for (I=0; I<N; I++)
```

```
    for (J=0; J<N; J++)
```

```
        if (I==J)
```

```
            U[I][J]=1;
```

```
        else
```

```
            U[I][J]=0;
```

```
/* Edition du résultat */
```

```
printf("Matrice unitaire de dimension %d :\n", N);
```

```
for (I=0; I<N; I++)
```

```
{
```

```
    for (J=0; J<N; J++)
```

```
        printf("%d", U[I][J]);
```

```
        printf("\n");
```

```
    }
```

```
return 0;
```

```
}
```

Programmation en langage C

Plan

1

Rappels (Types de bases, variables, opérateurs, structures de contrôles,..)

2

Les fonctions

3

Les pointeurs et l'allocation dynamique

4

Les chaines de caractères

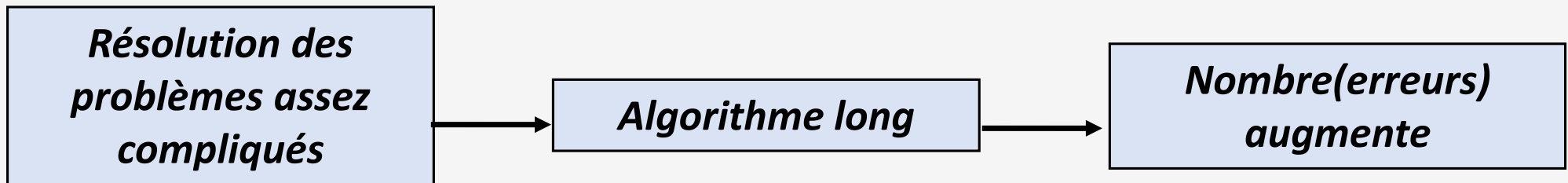
5

Les types composés (structures, unions, synonymes)

6

Les fichiers

- Certains problèmes conduisent à des programmes longs, difficiles à écrire et à comprendre. On les découpe en des parties appelées **sous-programmes** ou **modules**.
- Les modules sont des groupes d'instructions qui fournissent une solution à des parties bien définies d'un problème plus complexe. Ils ont plusieurs **intérêts**:
 - permettent de **"factoriser" les programmes**, càd de mettre en commun les parties qui se répètent
 - permettent **une structuration et une meilleure lisibilité** des programmes
 - **facilitent la maintenance** du code (il suffit de modifier une seule fois)
 - peuvent éventuellement être **réutilisées** dans d'autres programmes
- La structuration de programmes en sous-programmes se fait en C à l'aide des **fonctions**



Utilisation des procédures et fonctions

La phase d'analyse permet d'isoler différents sous-problèmes à résoudre, qui correspondront à des procédures ou fonctions utilisées ensuite pour la résolution du problème principal.

- Une fonction peut avoir autant de paramètres que l'on veut, même aucun (comme `void main(void)`)
- Une fonction renvoie une valeur ou aucune.
- Les variables déclarées dans une fonction sont locales à cette fonction
- L'imbrication de fonctions n'est pas autorisée:

une fonction ne peut pas être déclarée à l'intérieur d'une autre fonction. Par contre, une fonction peut **appeler** une autre fonction. Cette dernière doit être déclarée **avant** celle qui l'appelle.

2

Les Fonctions

Déclarer une fonction

TYPE de la valeur de retour
ex: int, float, int*,..., void

Les arguments et leurs types

<type de sortie> nom_fonction (type1 arg1,..., typeN argN)
{

Nom de la fonction

déclaration des variables internes de la fonction
instructions constituant le corps de la fonction

return (expression)

Valeur renvoyée

}

2

Les Fonctions

Déclarer une fonction

Exemples :

Une fonction qui calcule la somme de deux réels x et y :

```
double Som(double x, double y )  
{  
    return (x+y);  
}
```

Une fonction qui affiche la somme de deux réels x et y :

```
void AfficheSom(double x, double y)  
{  
    printf (" %lf", x+y );  
}
```

Une fonction qui renvoie un entier saisi au clavier

```
int RenvoieEntier( void )  
{  
    int n;  
    printf (" Entrez n \n");  
    scanf (" %d ", &n);  
    return n;  
}
```

Appeler une fonction

- L'appel d'une fonction se fait par simple écriture de son nom avec la liste des paramètres :
nom_fonction (para1,..., paraN)
- **Paramètre formel**: le paramètre lors de la déclaration d'une fonction.
- **Paramètre effectif** : le paramètre lors de l'utilisation de la fonction dans l'algorithme appelant.
- L'ordre et les types des paramètres effectifs doivent correspondre à ceux des paramètres formels

Le compilateur attend des doubles; les conversions sont automatiques

Ici, on ne tient pas compte de la valeur de retour

Exemple d'appels:

```
main( )
```

```
{
```

```
    double z;
```

```
    z=Som(2, 3);
```

```
    AfficheSom(6.5,7.2);
```

```
}
```


- Il est nécessaire pour le compilateur de connaître la définition d'une fonction au moment où elle est appelée. Si une fonction est définie après son premier appel (en particulier si elle est définie après main), elle doit être déclarée auparavant.
- La déclaration d'une fonction se fait par son **prototype** qui indique les types de ses paramètres et celui de la fonction :

type nom_fonction (type1,..., typeN);

2

Les Fonctions

Exemple

```
float calcule(float, float);
int addition();
void main(void)
{
    int Val ;

    Val = addition();
    printf("val = %d", Val);
}
float calcule(float A, float B)
{
    return ( (A+ B) / 2) ;
}
int addition()
{
    float tmp;
    tmp = calcule(2.5,3) + calcule(5,7.2);
    return (int)tmp;
}
```

- Le C est un langage structuré en blocs `{ }`, les variables ne peuvent être utilisées que là où elles sont déclarées
- On peut manipuler 2 types de variables dans un programme C : des **variables locales** et des **variables globales**. Elles se distinguent par ce qu'on appelle leur **portée** (leur "espace de visibilité", leur "durée de vie").
- Une variable définie à l'intérieur d'une fonction est une **variable locale, elle n'est connue qu'à l'intérieur de cette fonction**. Elle est créée à l'appel de la fonction et détruite à la fin de son exécution.
- Une variable définie à l'extérieur des fonctions est une **variable globale. Elle est définie durant toute l'application** et peut être utilisée et modifiée par les différentes fonctions du programme.

- Les variables déclarées au début de la fonction principale main ne sont pas des variables globales, mais elles sont locales à main.
- Une variable locale cache la variable globale qui a le même nom.
- Il faut utiliser autant que possible des variables locales. Ceci permet d'économiser la mémoire et d'assurer l'indépendance de la fonction.

2

Les Fonctions

Variables locales et globales

```
void maFonctionc(int x);  
int glo=0;           // variable globale  
void main(void)  
{  
    int    i = 5, j, k = 2; //locales à main  
    float  f = 2.8, g;  
    d = 3.7; ←  
    maFonction(i);  
}  
void maFonction(int v)  
{  
    double d, e = 0.0, f = v; //locales à fonction  
    i++; g--; glo++; ←  
    f = 0.0;  
}
```



Le compilateur ne connaît pas **d**



Le compilateur ne connaît pas **i** et **g**

C'est le **f** local qui est utilisé

- Les paramètres servent à **échanger des informations** entre la fonction appelante et la fonction appelée. Ils peuvent recevoir des données et stocker des résultats
- Il existe deux types principaux de passage de paramètres dans les langages de programmation :
 - **La transmission par valeur** : les valeurs des paramètres effectifs sont affectées aux paramètres formels correspondants au moment de l'appel de la fonction ou procédure. Dans ce mode le paramètre effectif ne subit aucune modification.
 - **La transmission par adresse (ou par référence)** : les adresses des paramètres effectifs sont transmises à la fonction appelante. Dans ce mode, le paramètre effectif subit les mêmes modifications que le paramètre formel.

- Pour effectuer une transmission par adresse en C, on déclare le paramètre formel de type pointeur et lors d'un appel de la fonction, on envoie l'adresse et non la valeur du paramètre effectif
- Exemple : `scanf()` travaille avec pointeur. C'est le pourquoi du `&`

2

Les Fonctions

Le passage des paramètres par valeur

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

```
void change(int v);
```

```
void main(void)
```

```
{
```

```
    int var = 5;
```

```
    change(var);
```

```
    printf("main: var = %d\n", var);
```

```
}
```

```
void change(int v)
```

```
{
```

```
    v *= 100;
```

```
    printf("change: v = %d\n", v);
```

```
}
```

change: v = 500
main: var = 5

2

Les Fonctions

Le passage des paramètres par valeur

Ecrire un programme utilisant une fonction qui calcule y tel que $y = x^2 + 5x + 6t^4$, x et t étant deux réels qcq.

```
float poly (float a, float b) /*passage des para par valeur*/
```

```
{float c ;
```

```
c=a*a+5*a+6*b*b*b*b ;
```

```
return(c) ;
```

```
}
```

Programme appelant :

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

```
int main()
```

```
{float x,t,y ;
```

```
float poly (float ,float);
```

```
printf("donner x et t ");
```

```
scanf("%f %f",&x,&t) ;
```

```
y=poly(x,t);
```

```
printf("le polynôme calculé pour %f et %f est %f \n",x,t,y) ;
```

```
return 0;}
```

Calcul de la moyenne d'un tableau

Fonction :

```
void moyenne (float tab[],int N,float * M)  /*passage par adresse*/  
{int i ;  
  *M=0 ;  
  for(i=0 ;i<=N-1 ;i++)  
    *M+=tab[i] ;  
    *M=*M/N;  
}
```

En C, le passage par adresse dans le sens propre n'est pas possible. Mais on peut y remédier grâce aux pointeurs, ie la fonction reçoit un pointeur pointant sur le paramètre correspondant.

Passage des paramètres par adresse

Programme appelant :

```
# include <stdio.h>
#define A 50
int main()
{float t[A],moy ;
int dim,i ;
void moyenne (float [],int,float *);
printf("donner la dimension du tableau\n") ;
scanf("%d",&dim) ;
for(i=0 ;i<=dim-1 ;i++)
{ printf("donner t[%d]\n",i) ;
scanf("%f",t+i) ;
}
moyenne (t,dim,&moy) ;    /*appel à la fonction*/
printf("la moyenne du tableau est %f \n",moy) ;
return 0;
}
```

- Une fonction qui fait appel à elle-même est une fonction **récursive**
- Dans une fonction récursive, il y a deux notions à retenir :
 - Quand la fonction s'appelle, on recommence avec de nouvelles données.
 - Il y a un test de fin, généralement en début de la fonction (il s'agit d'un cas limite (**cas trivial**) qui arrête la récursivité)

```
void F(...)  
{  
    if (fin) {  
        ...} else {  
            ...  
            F(...);  
            ...}  
}
```

Exemple 1 : Calcul du factorielle

La fonction factorielle d'un nombre entier positif :

$$\text{fact}(n) = n * (n-1) * (n-2) * \dots * 2 * 1 = n * \text{fact}(n-1)$$

La fonction fact est définie comme suit :

si $n=0$ alors $\text{fact}(n)=1$

si $n>0$ alors $\text{fact}(n)=n * \text{fact}(n-1)$

```
int fact (int n )
{
    if (n==0)  /*cas trivial*/
        return (1);
    else
        return (n*fact(n-1) ); /*appel récursif de la fonction*/
}
```

Remarque : l'ordre de calcul est l'ordre inverse de l'appel de la fonction

Exemple 1 : Calcul du factorielle**Remarque :**

Tout processus récursif peut être exprimé par son équivalent sous forme d'itérations.

La fonction *fact* aurait pu s'écrire de la manière qui suit :

```
int fact2(int n)
{ int i,f ;
  f=1 ;
  if (n!=0) for ( i=1 ;i<=n ;i++)
    f=f*i ;
  return (f);
}
```

L'avantage de la récursivité est qu'elle est plus naturelle et concise, elle économise l'écriture du code dans un programme. Son inconvénient est qu'elle occupe beaucoup d'espace mémoire à son exécution.

Exemple 2 : Fonction récursive qui calcule le terme n de la suite de Fibonacci définie par :

$$U(0)=U(1)=1$$

$$U(n)=U(n-1)+U(n-2)$$

```
int Fib(int n)
{
    if (n==0 || n==1)
        return (1);
    else
        return ( Fib(n-1)+Fib(n-2));
}
```

Une fonction itérative pour le calcul de la suite de Fibonacci :

```
int Fib (int n)
{
    int i, AvantDernier, Dernier, Nouveau;
    if (n==0 || n==1)
        return (1);
    AvantDernier=1; Dernier =1;
    for (i=2; i<=n; i++)
    {
        Nouveau= Dernier+ AvantDernier;
        AvantDernier = Dernier;
        Dernier = Nouveau;
    }
    return (Nouveau);
}
```

la solution
récursive est plus
facile à écrire



2

Les Fonctions

Passer des tableaux aux fonctions

- Les tableaux peuvent être passés comme paramètres d'une fonction.
- Ils ne peuvent pas être retournés comme résultat d'une fonction.
- La longueur du tableau ne doit pas être définie à la déclaration de la fonction.

```
#include <stdio.h>
void  somme(int x[], int n);
void  main(void){
    int i;
    int p[6] = { 1, 2,3, 5, 7, 11 };
    somme(p, 6);
    for (i=0;i<6;i++)
        printf(" %d ", p[i]);
}
void  somme(int a[], int n){
    int    i;
    for(i = n-1; i >0 ; i--)
        a[i] += a[i-1];
}
```

1 - Écrire une fonction récursive calculant la valeur de la « **fonction d'Ackermann** » "**Acker**" définie pour $m > 0$ et $n > 0$ par :

- $\text{Acker}(m,n) = \text{Acker}(m-1, \text{Acker}(m,n-1))$ pour $m > 0$ et $n > 0$
- $\text{Acker}(0,n) = n+1$ pour $n > 0$
- $\text{Acker}(m,0) = \text{Acker}(m-1,1)$ pour $m > 0$

Exercice 1 - correction

```
#include <stdio.h>
int acker (int, int);

int acker (int m, int n)
{
    if ( (m<0) || (n<0) )
        return 0;
    else if (m==0)
        return (n+1);
    else if (n==0)
        return (acker(m-1,1));
    else
        return acker(m-1, acker(m,n-1));
}

main ()
{
    int m,n;
    printf("donnez m et n :");
    scanf("%d %d", &m, &n);
    printf("acker(%d,%d)=%d", m, n, acker(m,n));
}
```

2 - Ecrire la fonction **NCHIFFRES** du type **int** qui obtient une valeur entière N (positive ou négative) du type **long** comme paramètre et qui fournit le nombre de chiffres de N comme résultat.

- Ecrire un petit programme qui teste la fonction NCHIFFRES:

Exemple:

Introduire un nombre entier : 6457392

Le nombre 6457392 a 7 chiffres.

Exercice 2 - correction

```
int NCHIFFRES(long N)
{
    int I;

    /* Conversion du signe si N est négatif */
    if (N<0)
    {
        N *= -1;
    }

    /* Compter les chiffres */
    for (I=1; N>10; I++)
    {
        N /= 10;
    }

    return I;
}
```

```
#include <stdio.h>
/* Prototypes des fonctions appelées */
int NCHIFFRES(long N);

main()
{
    /* Variables locales */
    long A;

    /* Traitements */
    printf("Introduire un nombre entier : ");
    scanf("%ld", &A);
    printf("Le nombre %ld a %d chiffres.\n", A, NCHIFFRES(A));

    return 0;
}
```