

Programmation Langage C

Youssef ALJ

26 mars 2020

- 1 Concepts de base
 - Phases de création d'un programme C
- 2 Variables et types de bases - entrée/sortie
 - Variables et types de base
 - Lecture et écriture
 - la fonction `printf`
 - la fonction `scanf`
- 3 Structures de contrôle
 - if ... else
 - Opérateurs de test
 - Division euclidienne
 - switch ... case
 - "if" et "else if"
 - De la logique
- 4 Les boucles
 - Boucle for
 - Boucle do while
 - Boucle while
 - Break et Continue
- 5 Tableaux en C
 - Déclaration et initialisation
 - Bonnes pratiques pour l'utilisation des tableaux
- 6 Les fonctions
 - Pourquoi utiliser les fonctions
 - Déclaration, définition, appel
 - Paramètres
 - L'expression return
 - Différents types de fonction

Pré-requis :

- Initiation au langage C (cf cours du premier semestre).
- Mathématiques de base (Terminale S : un peu d'arithmétique).

Organisation des notes :

- 15 % participation (compte rendu de TP).
- 15 % contrôle continu.
- 70 % examen

- Le langage C a été développé aux laboratoires AT&T dans les années 1970 par Dennis Ritchie.

Le cycle de création d'un programme en langage C est le suivant :

- 1 Concevoir un algorithme.
- 2 Utiliser un éditeur pour écrire le code source.
- 3 Compilation à partir du code source.
- 4 Édition des liens.
- 5 Éventuellement corriger les erreurs de compilation.
- 6 Exécuter le programme et le tester.
- 7 Éventuellement corriger les bugs.
- 8 Éventuellement Recommencer depuis le début.

On n'a pas besoin de mémoriser ces étapes. Ceci viendra avec la pratique.

Exemple : on veut écrire un algorithme qui affiche "bonjour tout le monde" à l'écran.

Début

Variables :

Écrire("Bonjour tout le monde")

Fin

- Avec un éditeur (par exemple notepad++) on saisit le texte suivant :

```
#include <stdio.h>

int main()
{
    printf("bonjour tout le monde");
    return 0;
}
```

- On sauvegarde (souvent avec les touches CTRL-s).
- Durant la sauvegarde, on fait attention à ce que le nom du fichier finisse par '.c' pour qu'il soit reconnu comme étant un programme C.
- Y a-t-il des mots qui vous sont familiers ?

Le compilateur est un programme spécial qui :

- lit les instructions enregistrées dans le code source "bonjour.c"
- analyse chaque instruction.
- traduit ensuite l'information en langage machine compréhensible par le micro-processeur de l'ordinateur.

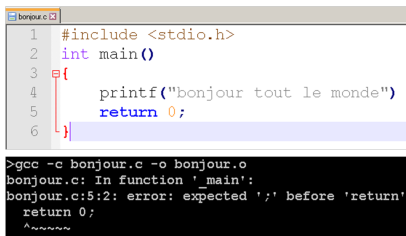
- Il existe plusieurs compilateurs :
 - gcc : GNU Compiler Collection.
GNU : acronyme récursif qui veut dire GNU's Not Unix.
 - Le compilateur : Microsoft Visual Studio.
 - ...
- On va utiliser le compilateur gcc. Voir TP.
- On compile en ligne de commandes windows (ou linux) via la commande suivante : `>gcc -c bonjour.c -o bonjour.o`
- Le résultat est la création d'un nouveau fichier appelé fichier objet qui a comme extension '.o' et qui a comme nom `bonjour.o`.

L'éditeur des liens (en anglais le linker) permet de :

- faire le liens entre les différents fichiers ".o"
- cette étape de link est aussi faite en utilisant `gcc`. `>gcc bonjour.o -o bonjour`
- on reviendra plus en détails sur le fonctionnement du linker dans les prochaines séances.

- Plusieurs compilateurs font la compilation et l'édition des liens en même temps.
- Cela se fait avec la commande suivante : `>gcc bonjour.c -o bonjour`


- Plusieurs erreurs de compilations peuvent apparaître.
- Elles sont dues par exemple à une mauvaise syntaxe (oubli du " ;", mot réservé mal écrit, etc.).
- Il faudra les corriger en éditant le fichier source, sauvegarder et recompiler puis ré-exécuter.



```
1  #include <stdio.h>
2  int main()
3  {
4      printf("bonjour tout le monde")
5      return 0;
6  }
```

```
>gcc -c bonjour.c -o bonjour.o
bonjour.c: In function '_main':
bonjour.c:5:2: error: expected ';' before 'return'
    return 0;
    ^~~~~~
```

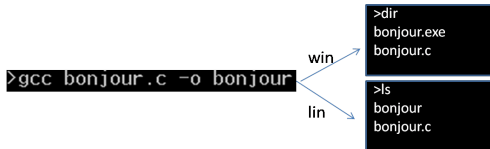
- Elles sont dues au fait que le linker n'arrive pas à faire le lien entre les fonctions définies par les programmes écrits.
- Une fonction principale doit toujours exister qui sert de point d'entrée aux autres fonctions.
- cette fonction principale est appelée la fonction `main`.



```
1 #include <stdio.h>
2 int _main()
3 {
4     printf("bonjour tout le monde");
5     return 0;
6 }
```



```
>gcc -c bonjour.c -o bonjour.o
>gcc bonjour.o -o bonjour
c:/mingw/bin/./lib/gcc/mingw32/6.3.0/../../../../libmingw32.a(main.o):(.text.startup+0xa0): undefined reference to `WinMain@16'
collect2.exe: error: ld returned 1 exit status
```



- Si la compilation s'est effectuée sans erreur on verra nouveau fichier qui apparait.
- Ce fichier est appelé un exécutable.
- Sous Windows, ce fichier exécutable porte l'extension '.exe' (voir figure ci-dessus).
- Pour lancer le fichier exécutable, dans notre exemple, on saisit en ligne de commande Windows : `>bonjour`.
- Dans un système d'exploitation de type Unix on saisit : `./bonjour`

- On utilise un programme appelé `gdb`.
- On reviendra sur la correction des bugs dans les prochaines séances.

Synthèse des principales étapes :

1

Edition

```
bonjour.c
1 #include <stdio.h>
2 int main()
3 {
4     printf("bonjour tout le monde");
5     return 0;
6 }
```

2

Compilation
+
Edition des liens

```
>gcc bonjour.c -o bonjour
```

win

lin

```
>dir
  bonjour.exe
  bonjour.c
```

```
>ls
  bonjour
  bonjour.c
```

3

Exécution

win

```
>bonjour
```

```
bonjour tout le monde
```

lin

```
>./bonjour
```

```
bonjour tout le monde
```

```
#include <stdio.h>
// ceci est un commentaire

int main()
{
    // un autre commentaire

    /*
    Une autre facon
    d ecrire un commentaire sur plusieurs lignes
    */
    printf("bonjour tout le monde");
    return 0;
}
```


- Les commentaires sont un outil de base pour documenter son programme.
- Cette documentation n'est pas utile seulement pour les autres mais pour vous aussi.
- Les commentaires améliorent la lisibilité du code écrit.
- Sans commentaire la lecture est difficile.
- Il est donc recommandé d'ajouter des commentaires autant que possible.

Toujours par souci de lisibilité il est recommandé respecter les règles d'indentation :

- Décaler d'un cran (= trois espaces) vers la droite tout bloc inclus dans un précédent.
- Cela permet de repérer qui dépend de quoi et si tous les blocs ouverts sont fermés.

Exemple d'un programme C mal écrit

```
#include <stdio.h>

void main()
{
    int a = 0;
    if (a == 0)

{
    printf("a est egal a 0");
}
else
{
    printf ("a est different de 0");
}
}
```

Exemple d'un programme C mieux écrit

```
#include <stdio.h>
// fonction main indispensable pour chaque programme C
void main()
{
    int a = 0;
    // si a est nul on affiche : "a est nul"
    if (a == 0)
    {
        printf("a est egal a 0");
    }
    // sinon on affiche : "a n est pas nul"
    else
    {
        printf ("a est different de 0");
    }
}
```

- Quand on programme avec n'importe quel langage de programmation on veut utiliser des données.
- On veut stocker ces données dans ce qu'on appelle des variables.
- On veut programmer un jeu vidéo.
- On a un joueur de ce jeu vidéo qui veut manipuler un personnage.
- Ce personnage va avoir une position dans une carte.
- Cette position va changer en fonction du mouvement du personnage.
- On veut stocker cette information pour pouvoir interagir avec d'autres éléments du jeu.

Il y en a six : `void`, `int`, `char`, `float`, `double`, `long double`.

- `void` : c'est le type vide. Il est surtout utilisé pour définir les fonctions sans arguments ou sans valeur de retour.
- `int` : c'est un type qui se décline sous plusieurs formes.
 - `short` : un entier court.
 - `long` : un entier long.
 - `signed` pour dire que l'entier qu'on manipule peut être positif ou négatif.
 - `unsigned` pour dire que l'entier qu'on manipule est uniquement positif**N.B. Si on n'indique rien, le qualificatif `signed` est appliqué.**

- `char` : ce type permet de stocker les caractères. Il peut aussi représenter les entiers sur 8 bits. On, peut aussi trouver un char signé `signed char` ou non signé `unsigned char`.
- `float` : ce type permet de représenter les réels.
- `double` : même chose que `float` mais avec une précision plus importante.
- `long double` : pareil que `double` mais avec une précision encore plus grande.

- La fonction `printf` est très utile car permet d'afficher des messages et les valeurs des variables.
- Exemple :

```
printf("Bonjour tout le monde!");  
printf("%d kilogramme vaut %d grammes", 1, 1000);
```

- L'argument de la fonction `printf` dit **format** est une chaîne de caractère qui détermine ce qui sera affiché par `printf` et sous quelle forme.
- Dans l'exemple 1 c'est "Bonjour tout le monde!".


```
printf("Bonjour tout le monde!");  
printf("%d kilogramme vaut %d grammes", 1, 1000):
```

- Dans l'exemple 2 c'est "%d kilogramme vaut
- Dans l'exemple 2 : la chaîne de caractère qu'on veut afficher est composée d'un texte normal et de séquence de contrôle permettant d'inclure des variables.
- le premier %d sera donc remplacé par la valeur 1.
- le deuxième %d sera remplacé par la valeur 1000.
- On aura comme résultat en sortie l'affichage suivant :
"1 kilogramme vaut 1000 grammes".

Les séquences de contrôle commencent par le caractère % suivi d'un caractère parmi les suivants :

- d ou i pour afficher un entier signé.
- f pour afficher un réel (float ou double).
- c pour afficher comme un caractère.
- s pour afficher une chaîne de caractères.

- Cette fonction permet de lire des données formatées à partir de l'entrée standard (clavier).
- `scanf` utilise les mêmes formats que `printf` mais on fait précéder le nom de la variable du caractère `&`.

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

- seul le format est passé en paramètre.
- Il ne faut ajouter de message ni aucun autre caractère.

```
/* Exemple pour tester "scanf" */
#include <stdio.h>
int main () {
    int nb1 ;
    float nb2 ;
    printf("Saisissez une valeur entiere (positive ou negative) pour nb1 : ") ;
    scanf("%d",&nb1) ;
    printf("Saisissez une valeur reelle pour nb2 : ") ;
    scanf("%f", &nb2) ;
    printf("nb1 vaut %d ; nb2 vaut %f\n", nb1, nb2) ;
    return 0;
}
```

- Éditer, sauvegarder puis compiler ce code.
- Faites en sorte que le programme affiche "hello monde" après avoir affiché les valeurs saisies par l'utilisateur.

- instruction1 n'est réalisée que si la condition est réalisée.

```
if (condition){  
    instruction1;  
}
```

```
void main()
{
    int var;
    printf("veuillez saisir un entier\n");
    scanf("%d", &var);
    if(var >= 0){
        printf("le nombre saisi est positif");
    }
}
```

```
if (condition){  
    instruction1;  
}  
else{  
    instruction2;  
}
```

- On exécute instruction1 si la condition est réalisée.
- Sinon on exécute instruction2.

```
void main()
{
    int var;
    printf("veuillez saisir un entier\n");
    scanf("%d", &var);
    if(var >= 0){
        printf("le nombre saisi est positif");
    }
    else{
        printf("le nombre saisi est negatif");
    }
}
```



```
if (condition1){  
    instruction1;  
}  
else if (condition2){  
    instruction2;  
}  
  
else if(condition3){  
    instruction3;  
}  
  
else{  
    instruction-par-defaut;  
}
```

- Si condition1 est vérifiée on exécute instruction1.
- Si condition2 est vérifiée on exécute instruction2.
- Si condition3 est vérifiée on exécute instruction3.
- Si aucune des conditions n'est vérifié alors on exécute instruction par défaut.

Opérateur	Signification
==	test d'égalité
!=	test de différence
>	supérieur
>=	supérieur ou égal
<	inférieur
<=	inférieur ou égal
&&	ET logique
	OU logique
!	NOT logique
^	XOR logique

A	B	$A \text{ ET } B$	$A \text{ ou } B$	$A \oplus B$
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Comment tester si deux nombres sont égaux avec l'opérateur XOR (en C ^)?

Comment tester si deux nombres sont égaux avec l'opérateur XOR (en C ^)?

```
#include<stdio.h>

int main()
{
    int x = 10;
    int y = 10;
    // pour deux nombres égaux l'opérateur xor renvoie zero
    // ici z vaut 0
    // int z = x ^ y;
    // printf("z=%d", z);
    if ( !(x ^ y) )
        printf(" x is equal to y ");
    else
        printf(" x is not equal to y ");
    return 0;
}
```

Que fait le programme suivant ?

```
// dans_le_mille.c
#define CIBLE.1 1000
#define CIBLE.2 100
#include <stdio.h>
#include <math.h>
int main() {
    float x, y;
    int danslemille, dehors, total_points=0;
    printf("x ? "); scanf("%f", &x); printf("x = %.2f\n", x);
    printf("y ? "); scanf("%f", &y); printf("y = %.2f\n", y);
    float d = sqrt(x*x + y*y);
    danslemille = (d < 1);
    dehors = (d > 3);
    if (danslemille) total_points = CIBLE.1;
    else if (!dehors) total_points = CIBLE.2;
    printf("total points = %d\n", total_points);
    return (0);
}
```

Théorème de la division euclidienne

Pour tout $(a, b) \in \mathbb{N} \times \mathbb{N}^*$, il existe un unique couple $(q, r) \in \mathbb{N} \times \mathbb{N}$ tel que :

$$a = bq + r \quad \text{avec } 0 \leq r < b$$

Exemple

Le reste de la division euclidienne de tout entier a par 2 est soit 0 soit 1.

Conséquence immédiate

- un entier a est pair si le reste de division euclidienne de a par 2 est 0. Il est impair sinon.

- le quotient de la division euclidienne d'un entier a par un entier b est obtenu avec l'opérateur `/`.
- le reste est obtenu avec `%`.


```
#include<stdio.h>

int main()
{
    int a,b,q,r;
    printf("veuillez saisir a\n");
    scanf("%d",&a);
    printf("veuillez saisir b\n");
    scanf("%d",&b);
    // calcul du quotient de a par b
    q = a/b;
    // calcul du reste de a par b
    r = a%b;
    printf("Le quotient est q=%d\n", q);
    printf("Le reste est r=%d\n", r);
    return 0;
}
```

- On veut vérifier si un nombre donné est multiple de 3.
- On veut vérifier si un nombre donné est pair ou pas.

- Une solution afin d'éviter les imbrications des instructions if.
- Si variable prend valeur1 on exécute instruction10 et instruction11.

```
switch( variable ){  
case valeur1 :  
    instruction10 ;  
    instruction11 ;  
    break ;  
case valeur2 :  
    instruction20 ;  
    instruction21 ;  
    break ;  
default :  
    instruction_par_defaut ;  
}
```

Pourquoi utiliser "else if" au lieu de plusieurs "if" ?

```
#include <stdio.h>
int main()
{
    int i;
    printf("veuillez saisir i\n");
    scanf("%d", &i);
    if(i > 0)
    {
        printf("i > 0\n");
    }
    if(i > 1)
    {
        printf("i > 1\n");
    }
    if(i > 2)
    {
        printf("i > 2\n");
    }
}
```

Ce programme va afficher :

veuillez saisir i

9

$i > 0$

$i > 1$

$i > 2$

```
#include <stdio.h>
int main()
{
    int i;
    printf("veuillez saisir i\n");
    scanf("%d", &i);
    if(i > 0)
    {
        printf("i > 0\n");
    }
    else if(i > 1)
    {
        printf("i > 1\n");
    }
    else if(i > 2)
    {
        printf("i > 2\n");
    }
}
```

Ce programme va afficher :

veuillez saisir i

9

$i > 0$

- On n'écrit pas : `if (12 <= x <= 14)`
- On écrit plutôt : `if ((12 <= x)&&(x <= 14))`

La boucle pour est constituée :

- d'initialisation exécuté **avant toutes les itérations**.
- de condition de boucle exécuté **avant chaque itération**.
- d'instruction de fin de boucle (souvent une incrémentation ou une décrémentation) exécuté **après chaque itération**.

```
for (initialisation ; condition ; incrementation)
{
    instructions_a_repeter ;
}
```

Exemple 1

```
#include <stdio.h>
int main()
{
    int i ;
    for (i = 0 ; i < 10 ; i = i + 1)
    {
        printf ("iteration %d \n", i) ;
    }
    printf ("valeur de i apres la boucle : %d \n", i) ;
    return 0 ;
}
```

On veut afficher les nombres pairs de 0 jusqu'à 10.

On veut afficher les nombres pairs de 0 jusqu'à 10.

```
#include <stdio.h>

int main()
{
    for (int i = 0; i <= 10; i++)
    {
        if (i%2 == 0)
        {
            printf("i=%d\n", i);
        }
    }
    return 0;
}
```

On peut initialiser la variable i à l'intérieur ou à l'extérieur de la boucle

```
for (int i = 0; i <= 10; i++)
{
    // instructions qu'on veut répéter
}
printf("i=%d\n", i);
```

ou

```
int i;
for (i = 0; i <= 10; i++)
{
    // instructions qu'on veut répéter
}
printf("i=%d\n", i);
```

Quelle est la différence entre les deux ?

On peut voir deux styles pour incrémenter dans la boucle.

```
for (int a =0; a<3; ++a)
{
    printf("a=%d\n", a);
}
```

ou

```
for (int a =0; a<3; a++)
{
    printf("a=%d\n", a);
}
```

Quelle différence y a-t-il ?

Qu'affiche le programme suivant ?

```
#include <stdio.h>

int main()
{
    int i=0,j=0;

    int k,l;
    k = ++i;
    l = j++;
    printf("i=%d\n",i);
    printf("j=%d\n",j);
    printf("k=%d\n",k);
    printf("l=%d\n",l);
    return 0;
}
```

- Sert à répéter des instructions dans lesquelles la condition est vérifiée à la fin.
- On est donc sûr d'exécuter au moins une fois le bloc d'instructions à répéter.

```
do {bloc d instructions a repeter;  
} while (condition de rebouclage) ;
```


Exemple de boucle "faire tant que"

```
#include <stdio.h>
int main () {
    int i = 0 ;
    do {
        printf ( "iteration %d \n", i ) ;
        i = i + 1 ;
    } while ( i < 10 ) ;
    printf ( "valeur de i apres la boucle : %d \n", i ) ;
    return 0 ;
}
```

- Dans la boucle while, le bloc peut ne jamais être exécuté.
- La condition est vérifiée avant le bloc.

```
while (condition de boucle) {  
    bloc d'instructions à répéter;  
}
```

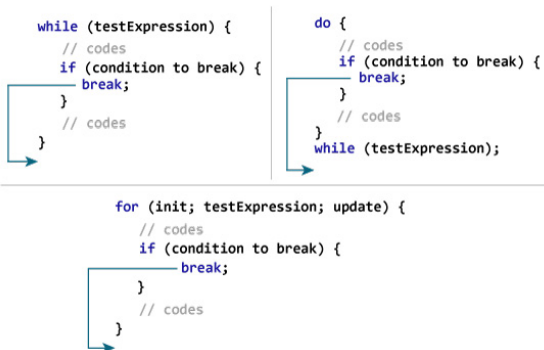
Boucle "tant que" exemple

```
#include <stdio.h>

int main () {
    int i = 0 ;
    while ( i < 10) {
        printf ( "iteration %d \n", i) ;
        i = i + 1 ;
    }
    printf ( "valeur de i apres la boucle : %d \n", i) ;
    return 0 ;
}
```

Attention : il ne faut pas oublier d'incrémenter *i*.

- L'instruction break permet de sortir d'une boucle immédiatement si la condition est réalisée.
- Surtout utilisable avec if et switch.



Qu'affiche le programme suivant?

```
# include <stdio.h>
int main()
{
    int i;
    double nombre, somme = 0.0;
    for(i=1; i <= 10; ++i)
    {
        printf("Entrer un nombre n%d: ", i);
        scanf("%lf", &nombre);
        // si nombre negatif saisi alors fin de la boucle
        if(nombre < 0.0)
        {
            break;
        }

        somme += nombre; // somme = somme + nombre;
    }
    printf("Somme = %.2lf", somme);
    return 0;
}
```

L'instruction continue permet de sauter l'itération courante de la boucle et continue avec la prochaine itération.

```
while (testExpression) {  
    // codes  
    if (testExpression) {  
        continue;  
    }  
    // codes  
}
```

```
do {  
    // codes  
    if (testExpression) {  
        continue;  
    }  
    // codes  
} while (testExpression);
```

```
for (init; testExpression; update) {  
    // codes  
    if (testExpression) {  
        continue;  
    }  
    // codes  
}
```

Qu'affiche le programme suivant?

```
# include <stdio.h>
int main()
{
    int i;
    double nombre, somme = 0.0;
    for(i=1; i <= 10; ++i)
    {
        printf("Entrer un nombre n%d: ", i);
        scanf("%lf",&nombre);

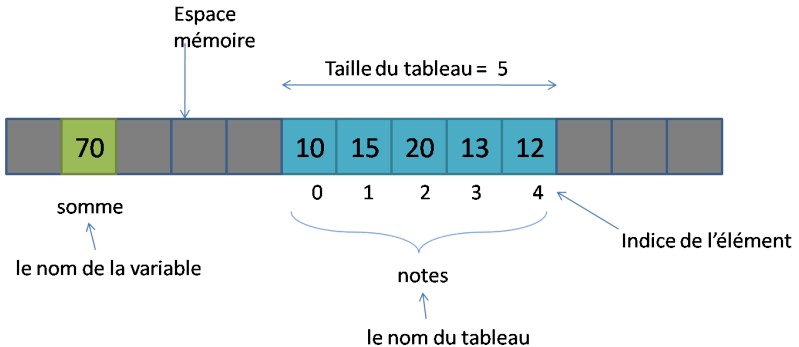
        if(nombre < 0.0)
        {
            continue;
        }

        somme += nombre; // somme = somme + nombre;
    }
    printf("Somme = %.2lf",somme);
    return 0;
}
```

- On veut un programme qui demande à l'utilisateur de saisir les notes de 100 étudiants et qui calcule leur moyenne.
- La solution naïve : il faut déclarer 100 variables de types `float` et calculer leur moyenne.
- TROP COMPLIQUÉ!!!
- Code trop long et très facile de commettre des erreurs.
- Une solution : utilisation de tableaux.

- Les tableaux en C sont représentés dans une zone continue de la mémoire (cf. slide suivant).
- Pour accéder à un élément particulier du tableau on utilise une valeur appelée **indice** (cf. slide suivant).
- l'indice d'un tableau commence par 0 et se termine par $N - 1$ où N est la taille du tableau (cf. slide suivant).
- Dans l'exemple du slide suivant l'indice du tableau varie de 0 à 4.

Exemple



```
type_de_donnees nom_du_tableau[TAILLE];
```

Avec :

- `type_de_donnees` : est un type de variable à choisir parmi `int`, `float`, `char` etc.
- `nom_du_tableau` : est un nom qu'on donne à notre tableau.
- `TAILLE` : est une constante qui définit la taille du tableau.

Exemple :

```
int notes[5];
```

On distingue deux types d'initialisation : initialisation statique et initialisation dynamique.

- Initialisation statique :

- On définit tous les éléments entre accolades pendant la déclaration.

```
int notes[5] = {10, 15, 20, 13, 12};
```

- On peut omettre la taille du tableau.

- Le compilateur détermine automatiquement la taille du tableau en utilisant le nombre d'éléments donné.

```
int notes[] = {10, 15, 20, 13, 12};
```

- Initialisation dynamique :

- On peut affecter aux éléments du tableau des valeurs dynamiquement c'est à dire à l'exécution du programme. Pour cela : on déclare d'abord un tableau et on utilise cette syntaxe :

- `nom_tableau[indice] = valeur;`

- Exemple : `notes[0] = 10;`

```
#include <stdio.h>

void main(){
    int indice;
    int notes[5];
    // on parcourt les elements du tableau
    for(indice = 0; indice < 5; indice++)
    {
        scanf("%d", &notes[indice]);
    }
}
```

Implémenter en C un programme qui demande à l'utilisateur de saisir 10 notes et qui calcule leur moyenne.

Implémenter en C un programme qui demande à l'utilisateur de saisir 10 notes et qui calcule leur moyenne.

```
#include <stdio.h>

// Une constante qui represente la taille du tableau
#define SIZE 10

void main(){

    int notes[SIZE];
    int indice , somme=0;
    float moyenne;
    for (indice = 0; indice<SIZE; indice++){
        scanf("%d", &notes[indice]);
        somme += notes[indice];
    }
    // on calcule la moyenne
    // on fait un cast : on convertit somme en float
    // le compilateur convertit automatiquement SIZE en float
    // le resultat sera affecte a moyenne
    moyenne = (float) somme/SIZE;
    printf("La moyenne des notes saisies est %.2f", moyenne);
}
```

N.B.

- **Faire attention quand on accède aux éléments d'un tableau.**
- Pb : L'accès à un élément qui n'existe pas ne génère pas d'erreur à la compilation.
- Le comportement à l'exécution est imprévisible :
 - On peut soit avoir des valeurs aléatoires.
 - ou un arrêt brusque du programme (program crash).

```
// ce programme compile correctement  
// mais affiche des valeurs bizarres  
// a l execution
```

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

```
int main()
```

```
{
```

```
    int arr[2];
```

```
    printf("%d ", arr[3]);
```

```
    printf("%d ", arr[-2]);
```

```
    return 0;
```

```
}
```


- On n'aura pas d'erreurs à la compilation de ce programme.
- On n'aura que des warning.
- Ce programme déclare un tableau de deux éléments.
- mais le remplit avec 5 éléments.

```
#include <stdio.h>
int main()
{
    // declaration de tableau avec initialisation
    // avec plus de valeurs que sa taille.
    int arr[2] = { 10, 20, 30, 40, 50 };
    for (int i=0; i<5;i++){
        printf("%d\n", arr[i]);
    }
    return 0;
}
```

La compilation de ce programme se fait sans erreur (seulement des avertissements) donc un exécutable est généré.

```
#include <stdio.h>
int main()
{
    // declaration de tableau
    // avec initialisation
    // avec plus de valeurs
    // que sa taille.
    int arr[2] = { 10, 20,
                  30, 40, 50 };
    for (int i=0; i<5;i++){
        printf("%d\n",arr[i])
    }
    return 0;
}
```

Résultat à la compilation :

```
ex4.c: In function 'main':
ex4.c:7:28: warning: excess elements in array initializer
      int arr[2] = { 10, 20, 30, 40, 50 };
                        ^
ex4.c:7:28: note: (near initialization for 'arr')
ex4.c:7:32: warning: excess elements in array initializer
      int arr[2] = { 10, 20, 30, 40, 50 };
                               ^
ex4.c:7:32: note: (near initialization for 'arr')
ex4.c:7:36: warning: excess elements in array initializer
      int arr[2] = { 10, 20, 30, 40, 50 };
                                   ^
ex4.c:7:36: note: (near initialization for 'arr')
```

Résultat à l'exécution :

```
10
20
2
0
0
```

Remarque 3 : une démonstration

```
// Un programme C qui montre que les elements d un tableau
// sont stockes dans des zones contigues en memoire

#include <stdio.h>
int main()
{
    // un tableau de 4 elements, si arr[0] est stocke
    // a l adresse x, alors arr[1] est stocke a x + sizeof(int)
    // arr[2] est stocke a x + 2*sizeof(int) etc
    int arr[5], i;
    printf("La taille d un entier en octet est %lu\n", sizeof(int));

    for (i = 0; i < 5; i++)
        // Notez l'utilisation de & pour acceder a ladresse
        // d une variable
        // utilisation de %p pour l affichage
        printf("Adresse de arr[%d] est %p\n", i, &arr[i]);

    return 0;
}
```

```
La taille d un entier est 4
Adresse de arr[0] est 000000000022FE30
Adresse de arr[1] est 000000000022FE34
Adresse de arr[2] est 000000000022FE38
Adresse de arr[3] est 000000000022FE3C
Adresse de arr[4] est 000000000022FE40
```

```
La taille d un entier est 4
Adresse de arr[0] est 000000000022FE30
Adresse de arr[1] est 000000000022FE34
Adresse de arr[2] est 000000000022FE38
Adresse de arr[3] est 000000000022FE3C
Adresse de arr[4] est 000000000022FE40
```

- Ce résultat montre que la taille (obtenue avec `sizeof()`) d'un entier en mémoire est 4 octets.
- Il montre aussi que les éléments d'un tableau sont stockés dans une zone contigues en mémoire. En effet on passe de l'adresse représentée par le nombre hexadécimal 22FE30 à l'adresse 22FE34.
- L'adresse d'une variable s'obtient avec l'opérateur `&`
- Pour afficher une adresse dans `printf`, on utilise le spécificateur de format : `%p`.
- Exemple :

```
int var;
printf("adresse de var en memoire est %p", &var);
```

Un programme C est un ensemble d'instructions qu'on veut exécuter.

Pour écrire un programme () C, on peut soit :

- Tout écrire dans un seul endroit, c'est à dire dans le `main()`. C'est ce qu'on faisait jusqu'à maintenant.
- Diviser le programme en plusieurs petites tâches (ou fonctions). Et utiliser la fonction `main()` pour exécuter ces tâches.

On veut savoir si un nombre est pair ou non

Version sans fonction

```
#include <stdio.h>
void main(){
    int a;
    printf("saisir entier\n");
    scanf("%d", &a);
    if (a % 2 == 0){
        printf("pair\n");
    }
    else{
        printf("impair\n");
    }
}
```

Version avec fonction

```
#include <stdio.h>
void affiche_parite(int a){
    if (a % 2 == 0){
        printf("pair\n");
    }
    else{
        printf("impair\n");
    }
}
void main(){
    int a;
    printf("saisir entier\n");
    scanf("%d", &a);
    affiche_parite(a);
}
```

- Une fonction = suite d'instructions groupées pour faire une tâche spécifique.
- Exemple de fonctions C déjà utilisées : `printf`, `scanf`, `sqrt`, etc.
- La plus importante reste la fonction `main` qui est indispensable pour chaque programme C.

Avantage 1

- **Ré-utilisabilité** : une fonction peut être définie une fois et ré-utilisée plusieurs fois.

Exemple : ré-utilisabilité

```
#include <stdio.h>
void affiche_parite(int x){
    if (x % 2 == 0){
        printf("pair\n");
    }
    else{
        printf("impair\n");
    }
}

void main(){
    int a,b,c;
    scanf("%d", &a);
    scanf("%d", &b);
    scanf("%d", &c);
    // utilisation 1
    affiche_parite(a);
    // utilisation 2
    affiche_parite(b);
    // utilisation 3
    affiche_parite(c);
}
```


Avantage 2 : Abstraction

- Abstraction : cacher les détails d'implémentation.
- Exemple : On utilise tous la fonction `printf()`.
- Mais on ne sait pas comment fait cette fonction réellement pour faire l'affichage.
- Ceci est un avantage car l'utilisateur de la fonction n'a pas besoin de savoir les détails de l'implémentation de `printf()` pour pouvoir l'utiliser.

Exemple : Abstraction

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

void main(){
    printf("la racine carree de 2 est %f",
        sqrt(2));
}
```

Avantage 3 : Débogage

- **Débogage** (en anglais debugging) :
- Dans l'exemple ci-contre si on n'avait pas utilisé la fonction `affiche_parite` on devrait faire un test de parité pour `a`, `b` et `c`.
- Si on s'aperçoit plus tard qu'il y a une erreur dans ce test il faudra corriger cette erreur 3 fois.
- avec l'utilisation de la fonction `affiche_parite`, on ne corrigerait cette erreur **qu'une seule fois**.

Exemple : débogage

```
#include <stdio.h>
void affiche_parite(int x){
    if (x % 2 == 0){
        printf("pair\n");
    }
    else{
        printf("impair\n");
    }
}

void main(){
    int a,b,c;
    scanf("%d", &a);
    scanf("%d", &b);
    scanf("%d", &c);
    // utilisation 1
    affiche_parite(a);
    // utilisation 2
    affiche_parite(b);
    // utilisation 3
    affiche_parite(c);
}
```

Il faut faire la distinction entre trois notions importantes concernant les fonctions : déclaration, définition et appel de fonction.

Définition de fonction

On définit le traitement de la fonction.

```
type_retour nom(params){  
    // code  
}
```

Déclaration de fonction

On dit au compilateur que la fonction existe.

Syntaxe :

```
type_retour nom( params );
```

Appel de fonction

On utilise la fonction.

Syntaxe : `nom(params);`

Exemple de définition

```
void affiche_parite(int x){  
    if (){  
        ...  
    }  
    else{  
        ...  
    }  
}
```

Exemple de déclaration

```
void affiche_parite(int x);
```

Exemple d'appel

```
affiche_parite(9);
```

Exemple 2 :

```
// exemple : calcul du produit de deux nombres
#include <stdio.h>

// declaration de la fonction calcul_produit
int calcul_produit(int x, int y);

// definition de la fonction main
void main(){
    // declaration des variables
    int a, b, produit;

    printf("saisissez deux entiers\n");
    scanf("%d%d", &a, &b);

    // appel de la fonction produit
    produit = calcul_produit(a,b);

    // affichage du resultat
    printf("Le produit est=%d", produit);
}

// definition de la fonction calcul_produit
int calcul_produit(int x, int y){
    int prod = x * y;
    return prod;
}
```

On distingue deux types de paramètres : formels et effectifs.

- Les paramètres formels : sont les paramètres avec lesquels la fonction est définie. **Ils sont utilisés avec leur type.**
- Les paramètres effectifs : sont les paramètres avec lesquels la fonction est effectivement appelée. **Ils sont utilisés sans leur type.**

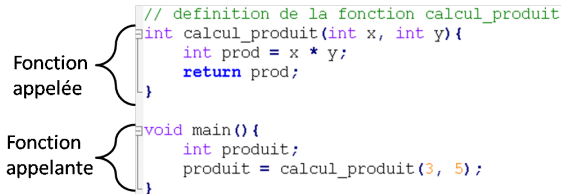
```
// definition de la fonction calcul_produit
int calcul_produit(int x, int y){
    int prod = x * y;
    return prod;
}

void main(){
    int produit;
    produit = calcul_produit(3, 5);
}
```

Paramètres formels

Paramètres effectifs

- Pour récupérer le résultat de calcul d'une fonction, on utilise l'expression `return valeur;` **dans la définition de la fonction.**
- On récupère ensuite le résultat de la fonction et on le stocke dans une nouvelle variable.



```
// definition de la fonction calcul_produit
int calcul_produit(int x, int y){
    int prod = x * y;
    return prod;
}

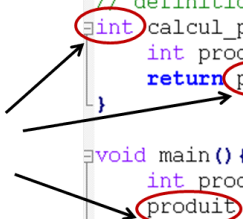
void main(){
    int produit;
    produit = calcul_produit(3, 5);
}
```

The diagram illustrates the relationship between a function definition and its call. A curly brace on the left groups the first function definition (`calcul_produit`) and is labeled "Fonction appelée" (Function called). Another curly brace on the left groups the `main` function, which calls `calcul_produit`, and is labeled "Fonction appelante" (Calling function).

Même type!!

```
// definition de la fonction calcul_produit
int calcul_produit(int x, int y){
    int prod = x * y;
    return prod;
}

void main(){
    int produit;
    produit = calcul_produit(3, 5);
}
```

The diagram consists of three black arrows originating from the text 'Même type!!' on the left. The first arrow points to the 'int' keyword in the function signature 'int calcul_produit'. The second arrow points to the 'return prod;' statement in the function body. The third arrow points to the 'produit' variable in the 'main' function's assignment statement 'produit = calcul_produit(3, 5);'. In the code, the 'int' in the function signature, 'return prod;', and 'produit' are circled in red to highlight that they all represent the same integer type.

- 1 Écrire une fonction appelée `max` qui prend en entrée deux entiers `a` et `b` et qui renvoie le maximum des deux.
- 2 Tester votre fonction dans une fonction `main` en demandant à l'utilisateur de saisir les valeurs de son choix.
- 3 Ajouter une nouvelle fonction `somme` qui calcule la somme des entiers de départ. La tester.
- 4 Ajouter une nouvelle fonction `moyenne` pour calculer la moyenne des deux entiers. La tester.


```
type_retour nom_de_la_fonction (parametres)
```

Cette syntaxe de la déclaration d'une fonction autorise 4 cas de figures possibles :

- Pas de type de retour et pas de paramètres.
- Pas de type de retour mais avec des paramètres.
- Avec un type de retour mais sans paramètres.
- Avec un type de retour et avec des paramètres.

Pas de type de retour et pas de paramètres.

Syntaxe

```
void nom_fonction()  
{  
    // corps de la fonction  
}
```

Avec un type de retour mais sans paramètres.

Syntaxe

```
type_retour nom_fonction()  
{  
    // corps de la fonction  
    return valeur;  
}
```

Pas de type de retour mais avec des paramètres.

Syntaxe

```
void nom_fonction(type1 param1, type2  
    param2, ...)  
{  
    // corps de la fonction  
}
```

Avec un type de retour et avec des paramètres

Syntaxe

```
type_retour nom_fonction(type1 param1,  
    type2 param2, ...)  
{  
    // corps de la fonction  
    return valeur;  
}
```

Exemple 1 : Pas de type de retour pas de paramètres

Pas de type de retour pas de paramètres

Syntaxe

```
void nom_fonction()  
{  
    // corps de la fonction  
}
```

Exemple

```
#include <stdio.h>  
// declaration  
void generer_paires();  
  
// definition de main  
void main(){  
    // appel  
    generer_paires();  
}  
  
// definition de generer-paires  
void generer_paires(){  
    int i;  
    for (i=0; i<100; i = i +2){  
        printf("%d\n", i);  
    }  
}
```

Pas de type de retour avec paramètres.

Syntaxe

```
void nom_fonction(type1 param1, type2  
    param2, ...)  
{  
    // corps de la fonction  
}
```

Exemple

```
#include <stdio.h>  
// declaration de affiche-parite  
void affiche_parite(int x);  
  
// definition  
void main(){  
    // appel  
    int a;  
    scanf("%d", &a);  
    affiche_parite(a);  
}  
  
// definition de affiche-parite  
void affiche_parite(int x){  
    if (x % 2 == 0){  
        printf("pair\n");  
    }  
    else{  
        printf("imppair\n");  
    }  
}
```

Exemple 3 : Avec type de retour pas de paramètres

Avec type de retour pas de paramètres

Syntaxe

```
type_retour nom_fonction()  
{  
    // corps de la fonction  
    return valeur;  
}
```

Exemple

```
#include <stdio.h>  
#include <stdlib.h> // utilise pour rand()  
  
// declaration de alea  
int alea();  
  
// definition main  
void main(){  
    int resultat;  
    // appel  
    resultat = alea();  
    printf("nb aleatoire=%d", resultat);  
}  
  
// definition de alea  
int alea(){  
    int a;  
    a = rand();  
    return a;  
}
```

Exemple 4 : Avec type de retour avec de paramètres

Avec type de retour avec de paramètres.

Syntaxe

```
type-retour nom-fonction(type1 param1,  
    type2 param2, ...)  
{  
    // corps de la fonction  
    return valeur;  
}
```

Exemple

```
#include <stdio.h>  
// declaration de pair_impair  
int pair_impair(int num);  
  
// definition main  
void main(){  
    int a, resultat;  
    scanf("%d", &a);  
    // appel  
    resultat = pair_impair(a);  
    if (resultat == 0){  
        printf("pair");  
    }  
    else{  
        printf("impair");  
    }  
}  
  
// definition de pair_impair  
int pair_impair(int x){  
    if (x % 2 == 0){  
        return 0;  
    }  
    else{  
        return 1;  
    }  
}
```