



UNIVERSITE PARIS-EST CRETÉIL  
IUT SENART-FONTAINEBLEAU  
DEPARTEMENT G.E.I.I.



## PROGRAMMATION EN LANGAGE C

BASES DE L'ALGORITHMIQUE

DOMINIQUE GARRIC  
[GARRIC@U-PEC.FR](mailto:GARRIC@U-PEC.FR)





## SOMMAIRE



### Objectif :

Apprendre les bases de la programmation algorithmique avec mise en œuvre en langage C.

### **Fiche 1 : PROGRAMMATION..... 5**

1.1. DEFINITIONS .....	7
a- Programme .....	7
b- Programmation .....	8
1.2. ALGORITHMES .....	9
a- Présentation des Algorithmes.....	9
b- Intérêt des Algorithmes .....	10
1.3. CODAGE EN LANGAGE C.....	10
ANNEXE : ANALYSE par approche descendante .....	11

### **Fiche 2 : DECLARATION VARIABLE..... 13**

2.1. DEFINITION .....	15
2.2. ALGORITHME (déclaration variable) .....	16
2.3. CODAGE EN LANGAGE C (déclaration variable).....	17

### **Fiche 3 : AFFICHAGE ECRAN..... 19**

3.1. ALGORITHME.....	21
3.2. CODAGE EN LANGAGE C.....	22

<b>Fiche 4 : SAISIE CLAVIER .....</b>	23
4.1. ALGORITHME.....	25
4.2. CODAGE EN LANGAGE C.....	26
<b>Fiche 5 : AFFECTATION.....</b>	27
5.1. ALGORITHME.....	29
5.2. CODAGE EN LANGAGE C.....	29
<b>Fiche 6 : DECLARATION CONSTANTE.....</b>	31
6.1. DEFINITION .....	33
6.2. ALGORITHME (déclaration constante) .....	34
6.3. CODAGE EN LANGAGE C (déclaration constante) .....	35
<b>Fiche 7 : MANIPULATION CARACTERE.....</b>	37
7.1. DEFINITION .....	39
7.2. ALGORITHME et C (déclaration variable CARACTERE) .....	39
7.3. ALGORITHME et C (INITIALISATION CARACTERE) .....	40
7.4. ALGORITHME et C (AFFICHAGE CARACTERE).....	41
7.5. ALGORITHME et C (SAISIE CARACTERE).....	42
a- Saisie d'une variable caractère (validation par RC).....	42
b- Saisie d'une variable caractère (à la volée).....	43
<b>Fiche 8 : CALCUL.....</b>	45
8.1. DEFINITION .....	47
8.2. ALGORITHME.....	48
8.3. CODAGE EN LANGAGE C.....	48
8.4. ATTENTION ! Typage des données & calculs.....	50
a- Définition division entière ou réelle .....	50
b- Erreurs de compatibilité de types.....	51

<b>Fiche 9 : ALTERNATIVE .....</b>	53
9.1. QUE SONT LES INSTRUCTIONS ALTERNATIVES ?.....	55
9.2. QUELS SONT LES OPERATEURS LOGIQUES ?.....	56
9.3. INSTRUCTION SI-SINON.....	58
a- Syntaxe et fonctionnement SI-SINON .....	58
b- Exemples avec l'instruction SI-SINON .....	59
9.4. INSTRUCTION SI-SINONSI.....	61
a- Syntaxe et fonctionnement SI-SINONSI.....	61
b- Exemples avec l'instruction SI-SINONSI.....	62
9.5. INSTRUCTION CHOIX-CAS.....	63
a- Syntaxe et fonctionnement CHOIX-CAS.....	63
b- Exemples avec l'instruction CHOIX-CAS .....	64
 <b>Fiche 10 : FONCTION STANDARD .....</b>	67
10.1. DEFINITION.....	69
10.2. ALGORITHME.....	70
10.3. CODAGE EN LANGAGE C .....	71
 <b>Fiche 11 : CREER FONCTION.....</b>	73
11.1. NECESSITE des FONCTIONS.....	75
11.2. creation de fonctions.....	78
11.3. fonctions et variables.....	79
11.4. fonctions AVEC PARAMETRES EN ENTREE .....	81
a- Syntaxe fonction avec Paramètre en Entrée.....	81
b- Exécution fonction avec Paramètre en Entrée.....	82
11.5. fonctions AVEC RESULTAT EN SORTIE.....	83
a- Syntaxe fonction avec Résultat en Sortie .....	84
b- Exécution fonction avec Résultat en Sortie .....	85
 <b>Fiche 12 : BOUCLE .....</b>	87
12.1. que sont les INSTRUCTIONS ITERATIVES ? .....	89
12.2. INSTRUCTION POUR .....	90
a- Syntaxe et fonctionnement POUR .....	90
b- Exemples avec l'instruction POUR .....	92
12.3. INSTRUCTIONS TANT QUE .....	93
a- Syntaxe et fonctionnement TANT QUE.....	93
b- Exemples avec l'instruction TANT QUE.....	95

<b>Fiche 13 : TABLEAU.....</b>	97
13.1. DEFINITION.....	99
13.2. ALGORITHME ET c (déclaration variable TABLEAU).....	101
13.3. INITIALISATION D'UNE VARIABLE TABLEAU .....	102
13.4. UTILISER UNE VARIABLE TABLEAU.....	104
a- Affichage d'un tableau.....	104
b- Stockage de valeurs dans un tableau .....	105
c- Calculer avec des valeurs stockées dans un tableau.....	106
<b>Fiche 14 : STRUCTURE.....</b>	107
14.1. DEFINITION.....	109
14.2. ALGORITHME ET c (déclaration TYPE STRUCTURE).....	109
14.3. ALGORITHME ET c (déclaration VARIABLE STRUCTURE) .....	111
14.4. ALGORITHME ET c (utilisation VARIABLE STRUCTURE).....	112
<b>Fiche 15 : POINTEUR .....</b>	113
15.1. DEFINITION.....	115
15.2. ALGORITHME ET c (déclaration variable POINTEUR) .....	116
15.3. INITIALISATION D'UNE VARIABLE POINTEUR .....	117
<b>Fiche 16 : FONCTION AVEC E/S.....</b>	119
16.1. fonction avec parametre en e/S (pointeur) .....	121
a- Syntaxe fonction avec Paramètre Pointeur .....	122
b- Exécution fonction avec Paramètre Pointeur .....	123
16.2. fonction avec parametre TABLEAU .....	124
a- Syntaxe fonction avec Paramètre Tableau.....	124
b- Exécution fonction avec Paramètre Tableau .....	126

# FICHE 1 : PROGRAMMATION





## PROGRAMMATION

Ressources :

[Diapos PROGRAMMER](#)



Objectif :

Définition de la programmation.

### 1.1. DEFINITIONS

Le programmeur crée des programmes par le processus de programmation.

#### a- Programme

Le dictionnaire Larousse précise : « *Un programme informatique est un ensemble d'instructions et de données représentant un algorithme qui est susceptible d'être exécuté par un ordinateur* ».

De façon générale, un **programme** définit des **traitements à appliquer à des informations**. Plus précisément, un **traitement** est décrit par une suite d'**instructions qui visent à manipuler, de façon ordonnée, des données numériques ou textuelles**. Une analogie peut être faite entre un programme et une recette de cuisine. La recette décrit le mode opératoire pour obtenir un plat, c'est-à-dire la liste et l'ordre des opérations à effectuer pour transformer les ingrédients, dont nature et quantité ont été préalablement précisées. Les opérations correspondent aux instructions et les ingrédients aux données.

Les programmes sont écrits dans des langages informatiques compréhensibles par les ordinateurs. Nous nous intéressons aux programmes écrits en **langage C**, qui est un langage algorithmique et procédural, à exécution séquentielle :

- Un **langage algorithmique** permet la résolution d'un problème informatique par analyse descendante (décomposition d'un problème en sous-problème jusqu'à descendre à des actions primitives -simples instructions-), voir la partie 1.4 de cette fiche.
- Un **langage procédural** organise ses traitements et données associées dans des sous-programmes appelés procédures ou fonctions.
- Une **exécution séquentielle** indique que l'ordinateur exécute les instructions élémentaires une par une dans l'ordre où elles se présentent et en suivant leur logique.

La famille des langages procéduraux/algorithmiques constitue la base de toute connaissance informatique en programmation. Cette approche permet d'appréhender les manipulations de base dans tous les programmes, telles que : calculer, choisir entre plusieurs cas, recommencer....

Les programmes peuvent prendre plusieurs formes, selon leur stade de développement et le formalisme dans lequel ils sont écrits :

- L'**Algorithm** est un programme en langage naturel (exemple : français) ;
- Le **Programme Source** est un programme en langage informatique (exemple : langage C) ;
- Le **Programme Exécutable** est un programme en langage binaire (directement interprétable par la machine).

## b- Programmation

Le dictionnaire Larousse précise : « *La programmation est l'ensemble des activités liées à la définition, l'écriture, la mise au point et l'exécution de programmes informatiques* ».

Le processus de **programmation** permet au programmeur de créer une **solution informatique exécutable** par un ordinateur (programme ou application), **qui répond à un cahier des charges**.

Les étapes de développement itératives sont, essentiellement :

1. **Lecture attentive et Etude du cahier des charges.**
2. **Conception de la solution** : Analyse par approche descendante du problème de façon à obtenir les **algorithmes** des fonctions (instructions ordonnées et données) : 45 % du temps de projet.
3. **Traduction** des algorithmes en **langage C** en respectant les **Règles de Qualité** et de **Documentation** du code source : 15 % du temps de projet.
4. **Tests** du programme pour tous les points du cahier des charges et dans tous les cas prévisibles : 40 % du temps de projet.

Remarque :

Entre les étapes de programmation 3 et 4, une étape est réalisée par l'ordinateur lorsque le programmeur choisit les opérations « Make » ou « Run » de l'environnement de développement intégré (EDI *Code::Blocks* ou *Visual C++ Express*, ...). Il s'agit de la traduction du code source en code binaire (étapes de **compilation** -compiling- et d'**édition de liens** -link-). Quelques éléments matériels permettant l'exécution des programmes doivent être connus :

- La **RAM** (*Random Access Memory*) : mémoire de travail de l'ordinateur, qui sert à charger temporairement toutes les applications en cours d'exécution, dont le programme que vous êtes en train d'exécuter (instructions et données codées en binaire). Cette mémoire est volatile : lorsque l'ordinateur n'est pas alimenté, elle est vidée.
- Le **CPU** (*Central Processing Unit*) : unité de traitement des programmes chargés dans la RAM. Il exécute les instructions une par une, selon leur ordre et leur logique d'exécution, et modifie les données, stockées dans la RAM, impliquées dans ces instructions. Le CPU contient l'**ALU** (*Arithmetic and Logic Unit*) qui est l'unité de calcul de l'ordinateur.

## 1.2. ALGORITHMES

Un **algorithme** précise, en LANGAGE NATUREL, les données et les instructions nécessaires, ainsi que l'enchaînement de ces dernières, afin de réaliser une tâche donnée. Les traitements sont structurés en différents blocs, appelés **fonctions**.

### a- Présentation des Algorithmes

Nous adopterons le formalisme suivant pour nos algorithmes :

```
// Commentaire d'en-tête d'algorithme
//-----
// Commentaire d'en-tête de fonction
ALGO Nom_Fonction1
    VAR      nomVariables : types // Commentaire de données
    CONST    NOMCONSTANTES=valeurs : types // Commentaire de données
DEBUT
    // Commentaire de partie
    Instruction1
    Instruction2

    // Commentaire de partie
    Instruction3
    Instruction4
FIN

// Commentaire d'en-tête de fonction
ALGO Nom_Fonction2
    VAR      nomVariables : types // Commentaire de données
    CONST    NOMCONSTANTES=valeurs : types // Commentaire de données
DEBUT
    // Commentaire de partie
    Instruction5
FIN
```

Remarque : les algorithmes peuvent aussi être représentés avec des graphiques, de type organigrammes :

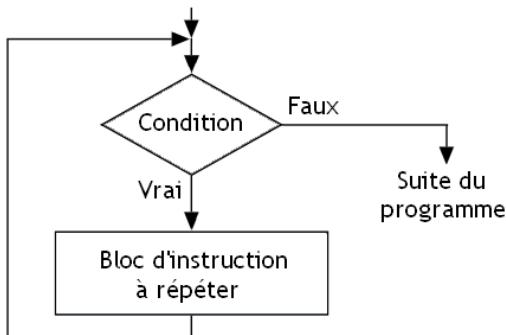


FIGURE 1 : ORGANIGRAMME DE LA BOUCLE TANT QUE

## b- Intérêt des Algorithmes

Les phases d'Analyse du cahier des charges et de Conception occupent une grande partie du temps de développement d'une application (autour de 45%) ; ce sont donc des phases importantes pour la construction des programmes : il ne suffit pas de se précipiter pour aligner des lignes de code. La conception comprend la création d'algorithmes prévisionnels du programme.

La réalisation préalable d'algorithmes permet de :

- se concentrer sur la recherche d'une solution logique ; en effet, en utilisant un langage naturel et simple, le programmeur s'affranchit du formalisme d'un langage informatique et ne bute donc pas sur la syntaxe pointilleuse du langage de programmation.
- planifier le développement par étape ; en effet, la réflexion préalable sur papier permet d'avoir une vue d'ensemble de la solution et d'éviter de recommencer un programme trop vite et mal parti !
- documenter le travail ; en effet, la traduction des algorithmes permet de commenter rapidement son code et de produire facilement, par la suite, des rapports de conception.

En conséquence, les **gains sur le processus de développement résultants du temps passé à établir des algorithmes** sont listés ci-dessous :

- Gain en temps et en qualité de développement global ;
- Gain en planification des tests ;
- Gain en efficacité de maintenance et de mise à jour.

## 1.3. CODAGE EN LANGAGE C

Un fichier source, écrit en langage C, est constitué (voir la Figure 2) :

- d'inclusions de bibliothèques : contenant les fonctions standards directement utilisables ;
- de fonctions successives (dont la principale est la fonction main()) : à l'intérieur d'une fonction, se trouvent d'abord la partie donnée, puis la partie traitement ;
- de commentaires de différents types : en-tête de fichier, en-tête de fonction, explications des données utilisées, indication du rôle de chaque partie.

```
/****************************************************************************
Commentaire d'en-tête de fichier source : Rôle fichier : ...      Auteur : ...      Date : ...
****************************************************************************/
#include <stdio.h>

// Commentaire d'en-tête de fonction
int main()
{   Instructions de déclaration de données ;           // Commentaire de données
    // Commentaire de partie
    Instruction1 ;
    Instruction2 ;
    // Commentaire de partie
    Instruction3 ;
    Instruction4 ;
}
```



La méthode pour approcher la solution progressivement consiste à partir d'un problème initial (cahier des charges) et à arriver à une solution finale (ensemble d'algorithmes directement traduisibles dans le langage de programmation) en passant par des niveaux intermédiaires, voir Figure 4, Figure 5 et Figure 6. Il s'agit de recommencer, tant que les sous-problèmes ne contiennent pas des instructions directement traduisibles :

- à décomposer les sous problèmes en nouveaux sous problèmes moins complexes ;
- à utiliser des exemples numériques pour aider à décomposer, pas à pas, le raisonnement.

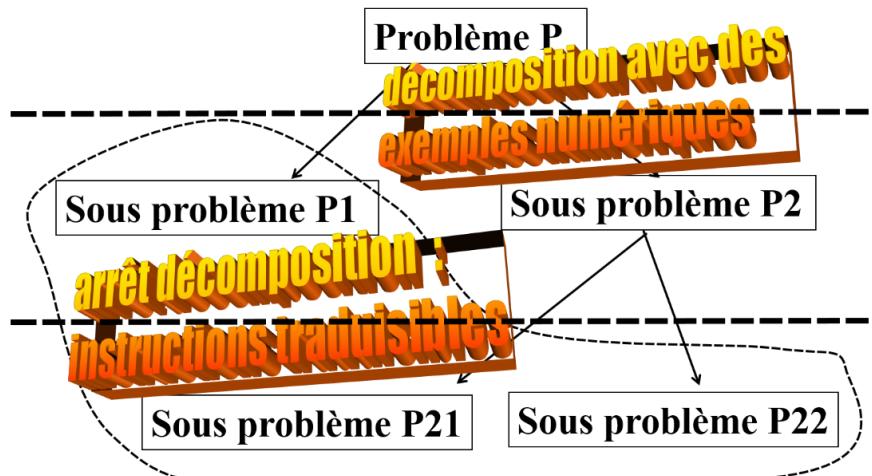


FIGURE 4 : ANALYSE D'UN PROBLEME PAR APPROCHE DESCENDANTE

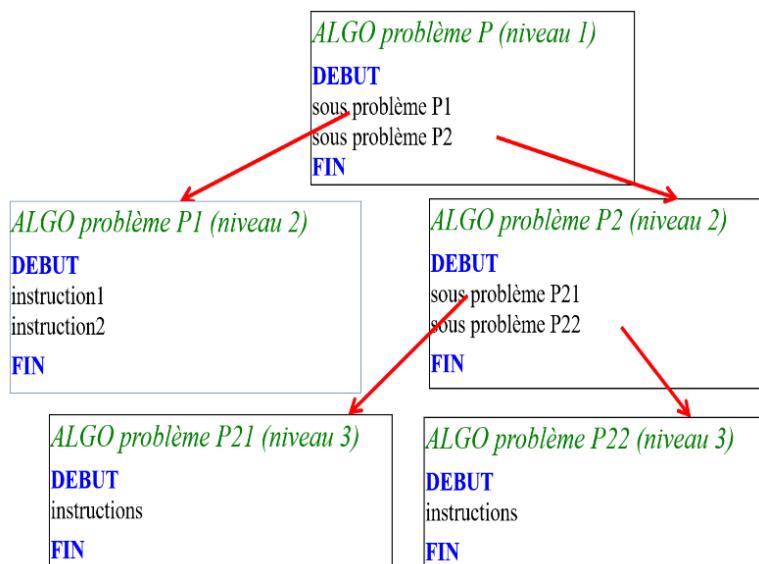


FIGURE 5 : ALGORITHMES OBTENUS

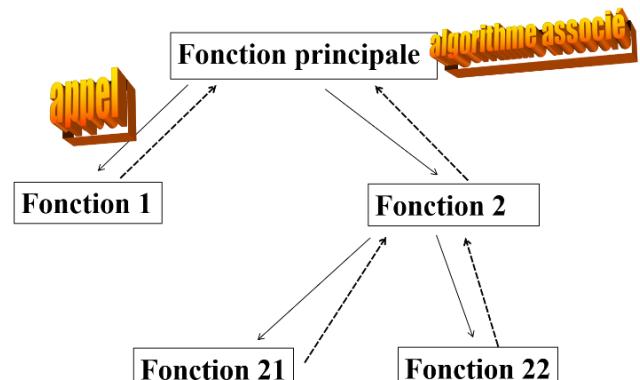


FIGURE 6 : FONCTIONS EN C OBTENUES

## FICHE 2 : DECLARATION VARIABLE





## DECLARATION VARIABLE

[Diapos MEMORISER](#)

### Objectif :

- ✓ Connaître la notion de *Variable*.
- ✓ Comment déclarer une variable dans un programme ?

### 2.1. DEFINITION

Une **variable** est une **donnée** qui sert à manipuler des grandeurs de diverses sortes (valeurs entières, réelles, caractères...) dans un programme. Les valeurs des variables sont **stockées dans la RAM** lorsque le programme est en cours d'exécution. La particularité d'une variable est que sa **valeur peut varier** en cours d'exécution du programme : elle peut être changée plusieurs fois par les instructions d'un programme.

Une variable, lors de l'exécution du programme, est décrite par :

- un **nom** (étiquette associée à une zone mémoire RAM -ex. *butsFr* sur le schéma en haut-) ;
- un **valeur** (contenu *3* dans la zone mémoire sur le schéma) ;
- une **adresse dans la RAM (pointeur *&butsFr* sur la zone RAM de la variable *butsFr*)**.

Pour caractériser une variable, il faut lui associer un **type de donnée**, qui indique quelle est sa nature ; il faut donc répondre aux questions suivantes pour choisir un type de donnée (voir Tableau pour l'ensemble des types simples du langage C) :

- sur combien de bits les valeurs de la variable sont-elles codées ?

Par exemple, une **taille de sa zone mémoire** de *2 octets* est associable à un type « petit entier » ou **entier court** et une taille RAM de 4 octets est associable à un type « grand entier » ou **entier long**.

- les valeurs de la variable commencent-elles à 0 ?

Ainsi, un entier naturel correspond à un **type non signé** (valeur minimum possible : 0) et un entier relatif correspond à un **type signé** (valeur minimum possible : une valeur négative).

- quel est le **domaine de capacité** de ma variable ?

Le domaine de capacité indique l'ensemble des valeurs que peut prendre une variable. Par exemple, une variable de type « entier non signé court » a des valeurs comprises dans [0,65535].

**⚠ ATTENTION aux DEPASSEMENT de CAPACITE (choix d'un type trop petit) !!**

- quels sont les opérateurs applicables à la variable ?

Par exemple : +, -, /, \* et % sont des opérateurs applicables à des entiers.

TYPE en C	Signification	Sigle en français	CAPACITE	TAILLE (en octets)
char	Caractère	c	[-128,127]	1
unsigned char	Caractère non signé	cns	[0,255]	1
unsigned short	Entier non signé court	ensc	[0,65 535]	2
short	Entier signé court	esc	[-32 768,32 767]	2
unsigned long unsigned int	Entier non signé long	ensl	[0,4 294 967 295]	4
long ou int	Entier signé long	esl	[-2147483648,2147483647]	4
float	Réel simple précision	rsp	[+- 3.4 $10^{38}$ , +- 3.4 $10^{38}$ ]	4
double	Réel double précision	rdp	[+-1.7 $10^{308}$ , +-1.7 $10^{308}$ ]	8

TABLEAU 1 : LES TYPES SIMPLES DU LANGAGE C

## 2.2. ALGORITHME (DECLARATION VARIABLE)

L'instruction de déclaration de variable permet d'indiquer les variables que le programme pourra manipuler. Elle s'écrit dans la partie données de la fonction qui utilise cette variable :

```
/*
***** DECLARATION DE VARIABLE *****
***** */
```

**ALGO DeclarerVariable // PARTIE DONNEES**

**VAR nomVariable : type // variables de type numérique ou caractère**  
 compt1 : ENTIER non signé court  
 compt2 : ensc  
 compt3, compt4 : unsigned short

**DEBUT**

**FIN**



### Règle de programmation : nommage des variables

- Noms explicites, pas trop longs;
- 1<sup>o</sup> lettre en minuscule (pour une variable locale, qui est déclarée dans une fonction et est une propriété privée de cette fonction);
- Commentaire explicatif pour préciser le rôle de la variable.

## 2.3. CODAGE EN LANGAGE C (DECLARATION VARIABLE)

La traduction en C consiste à indiquer d'abord **le type de la variable**, puis son nom suivi d'un ; :

```
/*****************************************************************************  
 * DECLARATION DE VARIABLE  
******/  
int main()  
{    // PARTIE DONNEES  
    unsigned short nomVariable1, nomVariable2; // TYPE numérique ou caractère  
    // PARTIE INSTRUCTIONS      ...  
}
```

L'exécution, par le CPU de l'instruction de déclaration de variable entraîne la réservation d'un espace mémoire vide (sans valeur), tel que (voir Figure 7) :

- sa longueur (nombre d'octets) dépend du type choisi ;
- son nom est *nomVariable* et son adresse est *&nomVariable*.

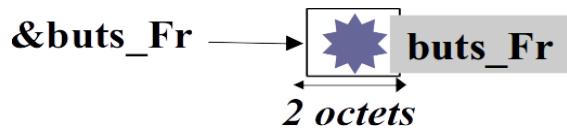


FIGURE 7 : EXECUTION CPU DE LA DECLARATION D'UN UNSIGNED SHORT



## FICHE 3 : AFFICHAGE ECRAN

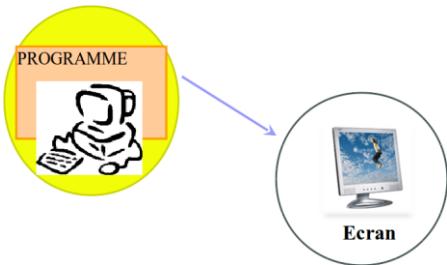




## AFFICHAGE ECRAN

Ressources :

[Diapos COMMUNIQUER](#)



Objectif :

- ✓ Comment afficher du texte et des données sur l'écran de l'ordinateur pour informer l'utilisateur de résultats ou pour lui demander des informations ?
- ✓ Comment gérer la disposition des informations sur l'écran ?

### 3.1. ALGORITHME

La **fonction AFFICHER()** permettra de donner un ordre d'envoi d'informations sur l'écran de l'ordinateur. Cette fonction est tout simplement paramétrée avec le texte à afficher sur l'écran ou/et avec le nom des données dont la valeur doit être montrée à l'utilisateur du programme. La gestion d'écran (disposition des informations sur l'écran) n'est pas forcément détaillée dans l'algorithme (quelques indications peuvent être, éventuellement, données) ; ce point sera à prendre en compte précisément lors de la traduction en C.

```
*****  
        AFFICHAGE DE TEXTE OU DE VALEURS DE DONNÉES SUR L'ECRAN  
*****  
ALGO AfficherEcran  
    VAR nomVariable1, nomVariable2 : type // variables de type numérique ou caractère  
DEBUT  
        // affichage texte seul  
        AFFICHER ("Texte à afficher sur l'écran...") au milieu de l'écran  
  
        // affichage valeur d'une variable  
        AFFICHER(nomVariable1)  
        // affichage valeurs de deux variables  
        AFFICHER(nomVariable1, nomVariable2)  
  
        // affichage de texte et de valeurs de variables intercalées dans le texte  
        AFFICHER("Il est possible d'indiquer les valeurs des données au milieu des phrases  
affichées sur l'écran, comme ici : la première variable vaut valeur 1 et la deuxième variable  
vaut valeur 2", nomVariable1, nomVariable2) sur une page effacée  
FIN
```

## 3.2. CODAGE EN LANGAGE C

La traduction en C consiste à :

- remplacer AFFICHER() par la fonction standard printf(),
- inclure la bibliothèque stdio.h, dans laquelle se trouve cette fonction standard,
- utiliser les %FORMAT pour indiquer l'endroit où est affichée une valeur dans le texte et pour préciser son format de donnée,
- rajouter la gestion d'écran.

Le CPU, exécutant un printf(), écrit les informations (texte et valeurs de variables récupérées dans la RAM) sur l'écran à l'endroit où se trouve le curseur d'écran.

```
/*************************************************************  
 * AFFICHAGE DE TEXTE OU DE VALEURS DE DONNÉES SUR L'ECRAN  
 ****/  
#include <stdio.h>  
  
int main()  
{    char, int, float, ...  nomVariable1, nomVariable2; // variables de type numérique ou caractère  
      // affichage texte seul à l'endroit où se trouve le curseur de l'écran  
      printf ("Texte à afficher sur l'écran...");  
      // affichage texte seul au milieu de l'écran (\n passe à la ligne, \t fait une tabulation)  
      printf ("\n\n\n\n\n\t\t\t\t\tTexte à afficher sur l'écran...");  
      // affichage valeur d'une variable selon son format, qui dépend de son type  
      printf(" %format",nomVariable1);  
      // affichage valeurs de deux variables selon leur format  
      printf(" %format1 %format2",nomVariable1, nomVariable2);  
      // affichage de texte et de valeurs de variables intercalées dans le texte  
      printf("Il est possible d'indiquer les valeurs des données au milieu des phrases affichées sur  
l'écran, comme ici : la première variable vaut %format1 et la deuxième variable vaut  
%format2",nomVariable1, nomVariable2);  
}
```

Les formats du printf() :

Format	Signification
%c	char
%hu	unsigned short
%hd	short
%lu	unsigned long
%f	float
%lf	double

Format	Signification
%hx	unsigned short affiché en Hexadécimal
%ld ou %d	long ou int
%lx	unsigned long affiché en Hexadécimal
.4f	float avec, au maximum, 4 décimales
.2lf	double avec, au maximum, 2 décimales

## FICHE 4 : SAISIE CLAVIER

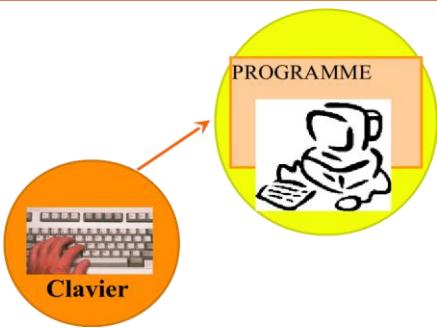




## SAISIE CLAVIER

Ressources :

[Diapos COMMUNIQUER](#)



Objectif :

Comment saisir des informations numériques ou textuelle que l'utilisateur du programme tape au clavier, afin de les affecter à des variables ?

### 4.1. ALGORITHME

La **fonction SAISIR()** est un ordre de récupération d'informations sur le clavier de l'ordinateur. Cette fonction est tout simplement paramétrée avec les adresses RAM des variables auxquelles doivent être stockées les valeurs tapées au clavier par l'utilisateur du programme. Un message doit être préalablement affiché sur l'écran, afin que l'utilisateur du programme comprenne ce qui lui est demandé.

```
/*************************************************************  
 * SAISIE DE VALEURS NUMERIQUES ou CARACTERES AU CLAVIER  
******/  
ALGO SaisirClavier  
    VAR nomVariable1, nomVariable2 : type //variables de type numérique ou caractère  
DEBUT  
    // demande de valeur à l'utilisateur du programme  
    AFFICHER("Donner la valeur de... : ")  
  
    // saisie de la valeur tapée au clavier (l'utilisateur doit finir en tapant sur la touche RC –entrée-)  
    SAISIR(&nomVariable1)  
  
    // demande de 2 valeurs à l'utilisateur du programme  
    AFFICHER("Donner la valeur de Variable1, puis la valeur de Variable2... : ")  
  
    // saisies des 2 valeurs tapées au clavier (l'utilisateur sépare avec ESPACE, ESCAPE ou RC)  
    SAISIR(&nomVariable1, &nomVariable2)  
FIN
```

## 4.2. CODAGE EN LANGAGE C

La traduction en C consiste à :

- remplacer SAISIR() par la fonction standard scanf(),
- inclure la bibliothèque stdio.h, dans laquelle se trouve cette fonction standard,
- utiliser les %FORMAT pour indiquer le type de la variable qui stocke la valeur tapée au clavier.

**Le CPU, exécutant un scanf(), attend de détecter l'appui sur la touche RC (ENTREE) du clavier. Une fois cette détection réalisée, le CPU récupère la valeur tapée avant RC, la convertit dans le type de donnée indiqué dans le %FORMAT, puis stocke la valeur convertie dans la zone RAM dont l'adresse a été fournie en paramètre du scanf().**

```
/*****************************************************************************  
 * SAISIE DE VALEURS NUMERIQUES ou CARACTERES AU CLAVIER  
******/  
#include <stdio.h>  
  
int main()  
{    int          nomVariable1; //variables de type numérique ou caractère  
      double       nomVariable2;  
  
      // demande de valeur à l'utilisateur du programme  
      printf("Donner la valeur de... : ");  
      // saisie de la valeur tapée au clavier (le CPU récupère après appui sur touche ENTREE)  
      scanf("%d", &nomVariable1);  
  
      // demande de 2 valeurs à l'utilisateur du programme  
      printf("Donner la valeur de Variable1, puis la valeur de Variable2... : ");  
      // saisies des 2 valeurs tapées au clavier (séparées par ESPACE, ESCAPE ou ENTREE)  
      scanf("%d%lf", &nomVariable1, &nomVariable2);  
}
```

Les formats du scanf() :

Format	Signification
%c	char
%hu	unsigned short
%hd	short
%ld ou %d	long ou int
%lu	unsigned long
%f	float
%lf	double

## FICHE 5 : AFFECTATION





## AFFECTATION

Ressources :

[Diapos MEMORISER](#)

&butsFr → 0 butsFr

Objectif :

Comment affecter (attribuer) une valeur à une donnée ?

### 5.1. ALGORITHME

L'**affectation** est une instruction qui permet d'**affecter (d'attribuer) une valeur à une donnée** ; elle s'écrit grâce à l'**opérateur d'affectation** **=**. L'affectation permet d'initialiser ou de modifier la valeur stockée dans la RAM à l'adresse de cette donnée. Il est possible d'affecter une valeur dans les parties données ou instructions du programme :

```
/*
***** AFFECTATION D'UNE VALEUR À UNE DONNÉE *****/
ALGO Affecter
    // affectation d'une valeur dans la partie données pour initialiser la variable
    VAR    nomVariable= 3 : type // variables de type numérique ou caractère
DEBUT
    nomVariable= 6 // affectation d'une nouvelle valeur dans la partie instructions
FIN
```

### 5.2. CODAGE EN LANGAGE C

La traduction en C est immédiate, car l'opérateur est le même **=**. Le CPU, exécutant une instruction d'affectation, attribue la valeur, qui est à droite du signe d'affectation **=**, dans l'espace mémoire de la variable placée à gauche du **=**. L'exécution d'une affectation entraîne une écriture dans la RAM (qui provoque l'écrasement de la valeur qui était préalablement stockée dans cet espace RAM).

```
/*
***** AFFECTATION D'UNE VALEUR À UNE DONNÉE *****/
int main()
{ int nomVariable= 3; // affectation d'une valeur dans la partie données pour initialiser la variable
  nomVariable= 6; // affectation d'une nouvelle valeur dans la partie instructions
}
```



## **FICHE 6 : DECLARATION CONSTANTE**





## DECLARATION CONSTANTE

&C → 300 000 C

Ressources :

[Diapos MEMORISER](#)

Objectif :

- ✓ Connaître la notion de *Constante*.
- ✓ Comment déclarer une constante dans un programme ?

### 6.1. DEFINITION

Il existe deux formes de **données** : les **variables** et les **constantes**. Les variables sont les plus couramment utilisées dans les programmes, car leur valeur peut changer autant de fois que nécessaire durant l'exécution du programme. Les constantes sont, dans certains cas, préférées aux variables.

Exemples de constantes:

TVA= 0.196

PI= 3.14

Une **constante** est une **donnée** dont la **valeur ne peut pas varier en cours d'exécution du programme** : sa valeur (de type entier, réel, caractère...) est fixée en début de programme, puis ne peut plus être changée par les instructions ; elle est **stockée dans la RAM en lecture seule** lors de l'exécution du programme.

Une constante, comme une variable, est décrite par :

- un **nom** (étiquette associée à une zone mémoire RAM -ex. C sur le schéma en haut-) ;
- un **valeur** (contenu 300 000 dans la zone mémoire sur le schéma) ;
- une **adresse dans la RAM (pointeur &C sur la zone RAM de la variable C)**.

Pour caractériser une constante, il faut lui associer un **type de donnée, compatible avec sa nature** ; il faut donc répondre aux questions suivantes (voir Tableau 2 pour l'ensemble des types simples du langage C) :

- sur combien de bits les valeurs de la variable sont-elles codées ?

Par exemple, une **taille de sa zone mémoire** de 2 octets est associable à un type « petit entier » ou **entier court** et une taille RAM de 4 octets est associable à un type « grand entier » ou **entier long**.

- les valeurs de la variable commencent-elles à 0 ?

Ainsi, un entier naturel correspond à un **type non signé** (valeur minimum possible : 0) et un entier relatif correspond à un **type signé** (valeur minimum possible : une valeur négative).

- quel est le **domaine de capacité** de ma variable ?

Le domaine de capacité indique l'ensemble des valeurs que peut prendre une variable. Par exemple, une variable de type « entier non signé court » a des valeurs comprises dans [0,65535].

**⚠ ATTENTION aux DEPASSEMENT de CAPACITE (choix d'un type trop petit) !!**

- quels sont les opérateurs applicables à la variable ?

Par exemple : +, -, /, \* et % sont des opérateurs applicables à des entiers.

## Les types simples du langage C :

TYPE en C	Signification	Sigle en français	CAPACITE	TAILLE (en octets)
char	Caractère	c	[-128,127]	1
unsigned char	Caractère non signé	cns	[0,255]	1
unsigned short	Entier non signé court	ensc	[0,65 535]	2
short	Entier signé court	esc	[-32 768,32 767]	2
unsigned long unsigned int	Entier non signé long	ensl	[0,4 294 967 295]	4
long ou int	Entier signé long	esl	[-2147483648,2147483647]	4
float	Réel simple précision	rsp	[+ - 3.4 $10^{38}$ , + - 3.4 $10^{38}$ ]	4
double	Réel double précision	rdp	[+ - 1.7 $10^{308}$ , + - 1.7 $10^{308}$ ]	8

TABLEAU 2 : LES TYPES SIMPLES DU LANGAGE C

## 6.2. ALGORITHME (DECLARATION CONSTANTE)

L'instruction de déclaration de constante permet d'indiquer les valeurs fixes que le programme pourra utiliser. Elle s'écrit dans la *partie données* de la fonction qui utilise cette constante :

```
*****
* DECLARATION DE CONSTANTE
*****
ALGO Declarerconstante // PARTIE DONNEES
    CONST NOMCONSTANTE= valeur : type // type numérique ou caractère
        PI= 3.14 : REEL simple précision // nombre PI pour calcul aire disque
DEBUT
FIN
```



### Règle de programmation : nommage des constantes

- Noms explicites et pas trop longs ;
- TOUTES les lettres en MAJUSCULES ;
- Commentaire explicatif pour préciser le rôle de la variable.

Dans son algorithme, le programmeur choisira une donnée constante, plutôt qu'une variable, lorsque la valeur de la donnée ne varie pas au cours de l'exécution du programme. Le **programmeur reconnaît visuellement qu'il s'agit d'une constante**, grâce aux majuscules, et ne risquera ainsi pas de tenter de modifier sa valeur dans ses instructions.

Le programmeur choisira une donnée constante, plutôt qu'une valeur écrite directement dans les instructions, car le **nom explicite d'une constante dans une instruction est plus lisible qu'une valeur** : la signification de la grandeur est directement compréhensible.

D'autre part, lorsqu'une valeur constante est utilisée plusieurs fois dans un programme, il est **plus simple de la déclarer une seule fois pour la maintenance du programme**. Si cette valeur change un jour (comme par exemple pour un taux de TVA qui est constant lors de l'exécution du programme, mais qui peut varier suite à une décision politique), le programmeur n'aura qu'à changer la valeur de sa constante, lors de sa déclaration, plutôt que toutes les instructions qui utilisent cette valeur.

Exemple d'utilisation de constante :       $\text{prixTTC} = \text{prixHT} \times (1 + \text{TVA})$

est plus lisible que :  $\text{prixTTC} = \text{prixHT} \times 1.196$

### 6.3. CODAGE EN LANGAGE C (DECLARATION CONSTANTE)

La traduction en C consiste à indiquer d'abord le **type de la constante, puis son nom suivi de sa valeur et d'un ;** :

```
/*****************************************************************************  
 * DECLARATION DE CONSTANTE  
******/  
  
int main()  
{ // PARTIE DONNEES  
    const TYPE      NOMCONSTANTE= valeur; // type numérique ou caractère  
    const float     PI= 3.14;           // nombre PI pour calcul aire disque  
  
    // PARTIE INSTRUCTIONS      ...  
}
```

L'exécution, par le CPU, de l'instruction de déclaration de constante entraîne la réservation d'un espace mémoire rempli (avec une valeur), tel que (voir Figure 8) :

- la longueur de l'espace mémoire (nombre d'octets) dépend du type choisi ;
- son nom est *PI* et son adresse est *&PI*.
- son contenu est 3.14 (affectation de la valeur à PI).

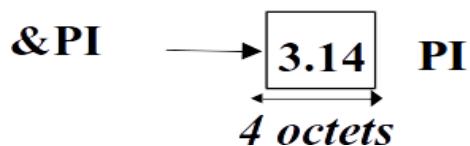


FIGURE 8 : EXECUTION D'UNE DECLARATION DE CONSTANTE

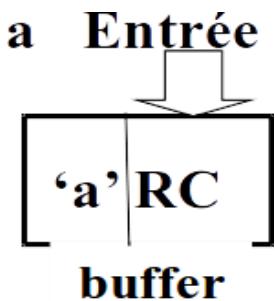


## FICHE 7 : MANIPULATION CARACTERE





## MANIPULATION CARACTERE



Ressources :

[Diapos MEMORISER](#)

Objectif :

- ✓ Connaître la notion de *Caractère*.
- ✓ Comment déclarer une variable de type caractère ?
- ✓ Comment afficher un caractère sur l'écran ?
- ✓ Comment saisir un caractère au clavier ?

### 7.1. DEFINITION

Le **type caractère** est un type simple qui permet de coder **toutes les touches du clavier** ; un caractère peut donc être une lettre, un chiffre, un signe de ponctuation, un caractère de contrôle non imprimable (CTRL...),.... Tout caractère est associé à un **code ASCII** (le code ASCII standard est un code entier sur 7 bits). Il existe deux types caractère en langage C (ou types d'entiers codés sur 1 octet) :

- **char** : caractère signé de valeurs entières comprises dans [-128,127] ;
- **unsigned char** : caractère non signé de codes ASCII compris dans [0,255].

La valeur d'un caractère est indiquée, dans un programme, par sa valeur entre simples guillemets (par exemple 'a') ou par sa valeur entière exprimée en base décimale (97<sub>10</sub>) ou hexadécimale (0x61). Un autre exemple de caractère est celui associé à la touche ENTREE : caractère RC (0xD ou 13<sub>10</sub>).

### 7.2. ALGORITHME ET C (DECLARATION VARIABLE CARACTERE)

L'**instruction de déclaration d'une variable de type caractère** permet d'indiquer les variables "caractère" que le programme manipulera. Elle s'écrit **dans la partie données de la fonction qui utilise cette variable** :

```
/*
***** DECLARATION DE VARIABLE CARACTERE *****
***** */

ALGO DeclarerVariableCarac // PARTIE DONNEES
  VAR      match : caractère non signé / caractère signé
           match : cns / c ou carac
           match : unsigned char / char
DEBUT
```



### Règle de programmation : nommage des variables de type caractère

- Noms explicites, pas trop longs;
- 1<sup>o</sup> lettre en minuscule (pour une variable locale, qui est déclarée dans une fonction et est une propriété privée de cette fonction);
- Commentaire explicatif pour préciser le rôle de la variable.

La traduction en C consiste à indiquer : **le type caractère de la variable, puis son nom suivi d'un ;**

```
/*****************************************************************************  
 * DECLARATION DE VARIABLE de TYPE CARACTERE  
******/  
  
int main()  
{    // PARTIE DONNEES  
    char  match;    // initiale du nom du match  
  
    // PARTIE INSTRUCTIONS      ...  
}
```

L'exécution, par le CPU de l'instruction de déclaration de variable caractère entraîne la réservation d'un espace mémoire vide (sans valeur), tel que :

- sa longueur est de 1 octet ;
- son nom est *match* et son adresse est *&match*.

## 7.3. ALGORITHME ET C (INITIALISATION CARACTERE)

L'initialisation d'une variable de type caractère se fait avec l'instruction d'**affectation** avec l'**opérateur d'affectation** =. La valeur affectée dans la RAM peut se présenter sous forme d'un caractère (entre simples guillemets) ou sous forme de la valeur entière d'un code ASCII. Il est possible d'affecter une valeur dans les parties données ou instructions du programme :

```
/*****************************************************************************  
 * INITIALISATION DE VARIABLE CARACTERE  
******/  
  
ALGO InitialiserVariableCarac  
// déclaration et initialisation de la variable caractère dans la partie données  
    VAR      match= 'A' : caractère  
DEBUT  
    match= 97 // le code ASCII 97 initialise la variable match avec le caractère associé  
FIN
```

La traduction en C est immédiate avec le même opérateur d'affectation =. Le CPU, exécutant une instruction d'affectation, attribue la valeur correspondant à un caractère, qui est à droite du signe d'affectation =, dans l'espace mémoire de la variable placée à gauche du =.

```
/*****************************************************************************  
 * INITIALISATION DE VARIABLE CARACTERE  
*****/  
int main()  
{   char   match= 'A';    // initiale du nom du match  
     match= 97;  
}
```

## 7.4. ALGORITHME ET C (AFFICHAGE CARACTERE)

La fonction **AFFICHER()** permettra de donner un ordre d'envoi d'informations sur l'écran de l'ordinateur. Cette fonction est tout simplement paramétrée avec le texte à afficher sur l'écran et avec le nom des données caractère dont la valeur doit être montrée à l'utilisateur du programme. La gestion d'écran (disposition des informations sur l'écran) n'est pas forcément détaillée dans l'algorithme.

```
/*****************************************************************************  
 * AFFICHAGE DE VARIABLE CARACTERE  
*****/  
ALGO AfficherVariableCarac  
  VAR      match= 'A' : caractère    // variables de type caractère  
DEBUT  
    AFFICHER("match : valeur", match)  
FIN
```

La traduction en C consiste à :

- remplacer **AFFICHER()** par la fonction standard **printf()** et inclure la bibliothèque **stdio.h**,
- utiliser les %FORMAT (caractère ou entier pour le code ASCII) et rajouter la gestion d'écran.

Le CPU, exécutant un **printf()**, écrit les informations (texte et valeurs de variables récupérées dans la RAM) sur l'écran à l'endroit où se trouve le curseur d'écran.

```
/*****************************************************************************  
 * AFFICHAGE DE VARIABLE CARACTERE  
*****/  
#include <stdio.h>  
int main()  
{   char match= 'A';    // initiale du nom du match  
     printf("Le nom du match est : %c",match); // affichage du caractère  
     printf("Le code ASCII du nom du match est : %d",match); // affichage du code ASCII- base 10  
     printf("Le code ASCII du nom du match est : %x",match); // affichage du code ASCII- base 16  
}
```

## 7.5. ALGORITHME ET C (SAISIE CARACTERE)

### a- Saisie d'une variable caractère (validation par RC)

La **fonction SAISIR()** est un ordre de récupération d'informations sur le clavier de l'ordinateur. Cette fonction est tout simplement paramétrée avec les adresses RAM des variables auxquelles doivent être stockées les valeurs caractère tapées au clavier par l'utilisateur du programme. Un message doit être préalablement affiché sur l'écran, afin que l'utilisateur du programme comprenne ce qui lui est demandé.

```
*****  
SAISIE CLAVIER DE VARIABLE CARACTERE, avec VALIDATION SAISIE par TOUCHE RC  
*****  
ALGO SaisirClavierCaracRC  
    VAR    match, match1, match2 : c // variables de type caractère  
DEBUT  
    // demande de valeur à l'utilisateur du programme  
    AFFICHER ("Donner nom match, match1 et match2 : ")  
    // saisie du caractère tapé au clavier (l'utilisateur doit finir en tapant sur la touche RC –entrée-)  
    SAISIR(&match)  
    // saisies de 2 caractères tapés au clavier (l'utilisateur sépare avec ESPACE, ESCAPE ou RC)  
    SAISIR(&match1, &match2)  
FIN
```

La traduction en C consiste à :

- remplacer SAISIR() par la fonction standard scanf(),
- inclure la bibliothèque stdio.h, dans laquelle se trouve cette fonction standard,
- utiliser le %c pour indiquer le type caractère de la variable qui stocke la valeur tapée au clavier.

Le CPU, exécutant un scanf(), attend de détecter la touche RC (ENTREE) dans la mémoire associée au clavier (buffer clavier ou *stdin* en langage C). Une fois cette détection réalisée, le CPU récupère le premier caractère disponible dans la mémoire du clavier, puis stocke ce caractère dans la zone RAM dont l'adresse a été fournie en paramètre du scanf().

```
*****  
SAISIE CLAVIER DE VARIABLE CARACTERE, avec VALIDATION SAISIE par TOUCHE RC  
*****  
#include <stdio.h>  
int main()  
{    char    match; // initiale du nom du match  
    printf("Donnez le nom du match : "); // demande de caractère à l'utilisateur du programme  
    scanf("%c" ,&match); // saisie valeur caractère du clavier (CPU récupère après détection de RC)  
}
```

Lorsque le CPU récupère des caractères dans le buffer clavier, ces caractères (et seulement ceux là) sont vidés du buffer ; le caractère RC, qui termine la saisie, n'est pas vidé du buffer. Ainsi, lorsque le buffer clavier contient déjà un caractère RC, le CPU ne se met pas en attente de l'appui sur la touche RC (il n'attend aucune nouvelle saisie clavier) : il récupère le caractère RC et le stocke à l'adresse de la variable, car RC est un caractère. Ce problème est alors illustré dans le programme ci-dessous (il faut donc avoir en tête que, lors de saisies numériques, le CPU enlève du buffer les caractères utilisés pour former les nombres, mais laisse le caractère de détection de fin de saisie, RC, dans le buffer) :

```
/****************************************************************************
 * PROBLEME SAISIE D'UN CARACTERE !!
 */
int main()
{
    char    match;      // initiale nom du match
    int butsFr, butsCr; // nombre buts de la France et de la Croatie

    printf("\nDonnez le score (France-Croatie)"); // saisie score du match
    scanf("%d%d", &butsFr, &butsCr);
    printf("Donnez le nom du match : "); // saisie initiale du nom du match
    scanf("%c", &match); //RC de la saisie numérique stocké dans match !!!!

}
```

Pour éviter des récupérations de caractères inappropriées, le programmeur doit effacer les caractères indésirables du buffer clavier, avec :

- **fflush(stdin)** : vide tous les caractères du buffer clavier (sous Windows, bibliothèque conio.h);
- **getchar()** : vide un caractère du buffer clavier (sous Linux et Windows, bibliothèque stdio.h).

## b- Saisie d'une variable caractère (à la volée)

La **fonction SAISIRVOLEE()** est un ordre de récupération d'informations sur le clavier de l'ordinateur. Cette fonction renvoie la valeur caractère lue au clavier, qui peut être stockée dans une variable par affectation. Un message doit être préalablement affiché sur l'écran, avant la saisie.

```
/*
 * SAISIE CLAVIER DE VARIABLE CARACTERE, A LA VOLEE
 */
ALGO SaisirVariableCaracVolee
    VAR      match, match1, match2: c
DEBUT
    // demande de valeur à l'utilisateur du programme
    AFFICHER ("Donner nom match, match1 et match2 : ")
    // saisie du caractère tapé au clavier dès qu'il est disponible (RC pas nécessaire)
    match= SAISIRVOLEE()
    // saisies de 2 caractères tapés au clavier (l'utilisateur les tape successivement sur le clavier)
    match1= SAISIRVOLEE()
    match2= SAISIRVOLEE()
FIN
```

La traduction en C consiste à :

- remplacer SAISIRVOLEE() par la fonction standard getche(),
- sous Windows : inclure la bibliothèque conio.h, dans laquelle se trouve cette fonction standard,
- sous Linux : recopier le code de la fonction getche() (voir plus bas).

Le CPU, exécutant un getche(), récupère le premier caractère disponible dans la mémoire du clavier ou, si *stdin* est vide, il attend que l'utilisateur tape un caractère au clavier. La valeur caractère finalement obtenue est stockée dans la zone RAM de la variable écrite devant le getche(). Lorsque le CPU récupère des caractères dans le buffer clavier, ces caractères (et ces caractères lus seulement) sont vidés du buffer. Toute touche du clavier correspond à un caractère qui pourra être affecté dans la variable caractère.

```
/*****************************************************************************  
 * SAISIE CLAVIER DE VARIABLE CARACTERE, A LA VOLEE, SOUS WINDOWS  
 ***************************************************************************/  
#include <stdio.h>  
#include <conio.h>  
  
int main()  
{    char    match; // initiale du nom du match  
    printf("Donnez nom match : ");  
    match= getche(); // caractère clavier directement récupéré sans attente touche RC  
}
```

```
/*****************************************************************************  
 * SAISIE CLAVIER DE VARIABLE CARACTERE, A LA VOLEE, SOUS LINUX  
 ***************************************************************************/  
#include <stdio.h>  
#include <stdlib.h>  
#include <termios.h>  
#include <unistd.h>  
  
int getche(void); /* reads from keypress, echoes */  
  
int main()  
{    char    match; // initiale du nom du match  
    printf("Donnez le nom du match : ");  
    match= getche(); // caractère clavier directement récupéré sans attente touche RC  
}  
  
int getche(void) /* reads from keypress, echoes */  
{    struct termios oldattr, newattr;  
    int ch;  
    tcgetattr( STDIN_FILENO, &oldattr );  
    newattr = oldattr;  
    newattr.c_lflag &= ~( ICANON );  
    tcsetattr( STDIN_FILENO, TCSANOW, &newattr );  
    ch = getchar();  
    tcsetattr( STDIN_FILENO, TCSANOW, &oldattr );  
    return ch;  
}
```

# FICHE 8 : CALCUL

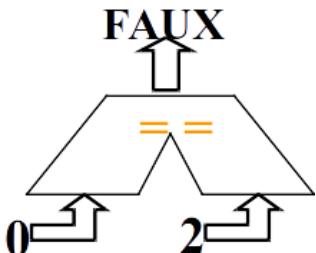




## CALCUL

Ressources :

[Diapos CALCULER](#)



Objectif :

- ✓ Quels calculs peuvent être réalisés dans un programme ?
- ✓ Comment écrire une instruction de calcul ?
- ✓ Comment le CPU exécute-t-il une instruction de calcul ?

### 8.1. DEFINITION

A partir de données en Entrée (valeurs en général saisies au clavier ou initialisées par affectation), un programme effectue un traitement et produit des données en Sortie (résultats en général affichés sur l'écran). Ce traitement est souvent un **calcul** sur les données d'Entrée.

Un calcul s'écrit à partir d'une **formule de calcul**, qui peut être arithmétique, binaire ou logique :

- une formule arithmétique utilise des opérateurs arithmétiques pour effectuer des calculs en base décimale (addition, soustraction, multiplication, division, modulo) ;
- une formule binaire utilise des opérateurs binaires pour effectuer des calculs en base 2 (inverseur, ET binaire, OU binaire, OU exclusif) ;
- une formule logique utilise des opérateurs logiques (de comparaison -supérieur, inférieur, égal, différent- et booléens -NON logique, ET logique, OU logique-). Les équations logiques, utiles pour les conditions logiques de certaines instructions, sont vues dans la partie traitant des alternatives.

Les **formules arithmétiques permettent d'écrire les calculs courants en base 10** (travaillant sur des **entiers ou des réels**) ; elles peuvent-être composées :

- de **fonctions mathématiques standards** : fonctions trigonométriques -ex. fonction cosinus cos(theta)- ou arithmétiques -ex. fonction racine carrée sqrt(a)- de la bibliothèque standard *math.h*.
- d'**expressions arithmétiques** construites avec des **opérateurs arithmétiques**, voir Tableau 3.

En ALGO	Nom	En C
+	Addition	+
-	Soustraction	-
x	Multiplication	*
/	Division	/
%	modulo	%

TABLEAU 3 : OPERATEURS ARITHMETIQUES

Exemple d'expressions arithmétiques :

- $(axb+3.) / (c-5.)$  : **les points, après les valeurs, indiquent que les valeurs sont de type réel.**
- $a \% b$  : le modulo donne le reste de la division euclidienne de a par b (**a et b sont des entiers !**).

## 8.2. ALGORITHME

Une instruction de calcul est une instruction qui permet de demander au CPU de faire effectuer un calcul par l'UAL (Unité Arithmétique et Logique). Elle s'écrit à partir d'une affectation car le résultat du calcul doit être affecté à une variable résultat dans la RAM. La **variable résultat doit toujours être avant le signe d'affectation = ; une formule de calcul se trouve à droite du signe d'affectation :**

```
=====
INSTRUCTION DE CALCUL
=====

ALGO Calculer
  VAR      a=5.3, b= 2., r=18.5 : réels double précision // opérandes de calcul
                                moy, aire : réels double précision // moyenne des réels et aire du disque
  CONST    PI= 3.14 : réel double précision
DEBUT
  // nomVariableResultat= Formule de Calcul
  moy= (a+b)/2.           // moyenne arithmétique de a et b
  aire= PI * r2          // aire du disque de rayon r
FIN
```

## 8.3. CODAGE EN LANGAGE C

La traduction en C est immédiate, il faut adapter les opérateurs et utiliser les fonctions standards avec la bibliothèque mathématique :

```
=====
INSTRUCTION DE CALCUL
=====

#include <math.h>

int main()
{
    double a=5.3, b= 2., r=18.5; // opérandes de calcul
    double moy, aire; // moyenne des réels et aire du disque
    const double PI= 3.14;

    // nomVariableResultat= Formule de Calcul
    moy= (a+b)/2.;           // moyenne arithmétique de a et b
    aire= PI * pow(r,2.);    // aire du disque de rayon r
}
```

Le CPU, exécutant une instruction de calcul, réalise deux étapes :

1. Il fait d'abord effectuer le calcul par l'UAL (formule de calcul à droite du signe d'affectation) ; les valeurs utiles pour le calcul sont fournies à l'UAL, qui évalue alors le résultat du calcul.
2. Il attribue, ensuite, la valeur du résultat fourni par l'UAL, à l'espace mémoire de la variable placée à gauche de l'opérateur d'affectation = (valeur résultat affectée à la variable résultat). Pour rappel : l'exécution de l'affectation entraîne une écriture dans la RAM (qui provoque l'écrasement de la valeur qui était préalablement stockée dans l'espace RAM).

L'exemple de la décomposition de l'exécution de l'instruction de soustraction  $a-b$  est donné en Figure 9 :

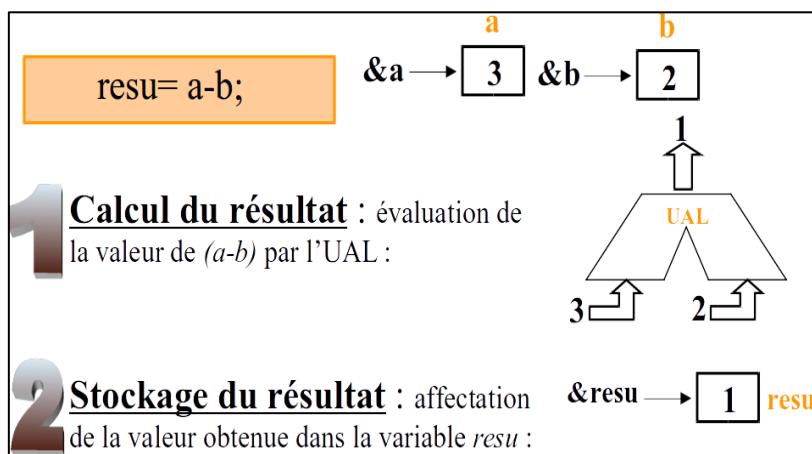


FIGURE 9 : EXECUTION DE LA SOUSTRACTION (A-B) PAR LE CPU

Concernant la première partie, pendant laquelle l'UAL évalue le calcul, les **règles d'évaluation** classique suivantes sont appliquées successivement :

1. L'ordre de calcul dépend des **parenthèses** (elles forcent les priorités). Ex. :  $(4+2)x2$  vaut 12.
2. L'ordre de calcul dépend, ensuite, de la **priorité des opérateurs**. Ex. :  $4+2x2$  vaut 8. Le Tableau 4 précise l'ordre de priorité de l'ensemble des opérateurs du langage C.
3. Pour un même niveau de priorité, l'évaluation de l'expression se fait de **gauche à droite** (à partir de l'opérateur d'affectation =). Ex. :  $4/2/2$  vaut 1.

Priorité	Opérateur
1	$()$
2	<b>NON</b> ( <i>non logique</i> ) — ( <i>complément à 1</i> )
3	<b>X</b> ( <i>multiplication</i> ) / ( <i>division</i> ) % ( <i>modulo</i> )
4	+ ( <i>addition</i> )      - ( <i>soustraction</i> )
5	<                  > $\leq$ $\geq$
6	<b>==</b> ( <i>égalité</i> ) <b><math>\neq</math></b> ( <i>différent</i> )
7	. ( <i>et binaire</i> )
8	$\oplus$ ( <i>ou exclusif</i> )
9	+ ( <i>ou binaire</i> )
10	<b>ET</b> ( <i>et logique</i> )
11	<b>OU</b> ( <i>ou logique</i> )
12	<b>=</b> ( <i>affectation</i> )

TABLEAU 4 : ORDRE DE PRIORITE DES OPERATEURS

**ATTENTION à ne pas confondre les opérateurs = (affectation) et == (égalité) !!**

- = permet d'écrire dans la RAM (une valeur est attribuée à une donnée)
- == permet de lire dans la RAM (deux valeurs sont comparées)

## 8.4. ATTENTION ! TYPAGE DES DONNEES & CALCULS

Des erreurs de calcul peuvent se produire avec la division si le programmeur ne fait pas attention aux types de données utilisés. Le programmeur écrit a/b dans son code que ce soit pour faire une division entière (division euclidienne avec reste et dividende) ou une division réelle (résultat avec une partie décimale). Il doit donc savoir ce qui indique au CPU et à l'UAL la division à réaliser (entière ou réelle).

### a- Définition division entière ou réelle

Donc, en écrivant a/b, le CPU va-t-il réaliser une division entière ou réelle ? La règle est :

- Si **a ET b sont des entiers**, l'UAL effectue la division entière de a par b. Le résultat du calcul est :
  - ✓ le quotient, entier résultat de l'application de l'opérateur / (*division*),
  - ✓ le reste, entier résultat de l'application de l'opérateur % (*modulo*).

```
*****  
EXEMPLE DE DIVISION ENTIERE  
*****  
int main()  
{      int      a=3, b=2; // opérandes entiers  
        int      div, reste; // quotient et reste  
        // division entière  
        div= a / b;  
        reste= a % b;  
}
```

#### Exemple de division entière

$$\begin{array}{ccc} a (3) & & b (2) \\ \text{reste (1)} & \text{---} & \text{div (1)} \end{array}$$

- Si **a OU b est un réel**, l'UAL effectue la division réelle de a par b. Le résultat du calcul est un nombre décimal, réel résultat de l'application de l'opérateur / (*division*).

```
*****  
DIVISION REELLE  
*****  
int main()  
{      float    a=3., b=2.; // opérandes réelles  
        float    div; // résultat de la division réelle  
        // division  
        div= a / b;  
}
```

#### Exemple de division réelle

$$\begin{array}{l} a (3.) / b (2.) \\ \text{donne : div (1.5)} \end{array}$$

## b- Erreurs de compatibilité de types

Les exemples suivants illustrent des erreurs de compatibilité de types lorsque le programmeur mélange les types de données dans les calculs :

- Les opérandes sont entiers et le résultat est réel : si le programmeur espérait obtenir le résultat de la division réelle, c'est raté ! La division entière est réalisée car a et b sont entiers ; le résultat a une décimale nulle, puisque la valeur finale est convertie en réel, voir **Erreur ! Source du renvoi introuvable.** :

```
*****  
ERREUR TYPES 1  
*****  
int main()  
{    int      a=3, b=2;      // opérandes entiers  
        float   div;          // résultat réel  
  
        div= a / b;          // division ENTIERE  
}
```

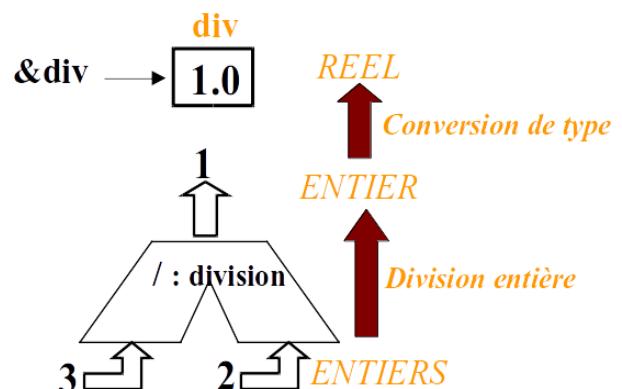


FIGURE 10 : DIVISION ENTIERE & AFFECTATION DANS REEL

- Les opérandes sont réels et le résultat est entier : si le programmeur espérait obtenir le résultat de la division réelle, c'est raté ! La division réelle est bien réalisée car a ou b sont réels ; par contre, le résultat est tronqué (partie décimale supprimée), puisque la valeur finale est convertie en entier, voir Figure 11 :

```
*****  
ERREUR TYPES 2  
*****  
int main()  
{    float   a=3., b=2.;    // opérandes réels  
        int     div;          // résultat entier  
  
        div= a / b;          // division REELLE  
}
```

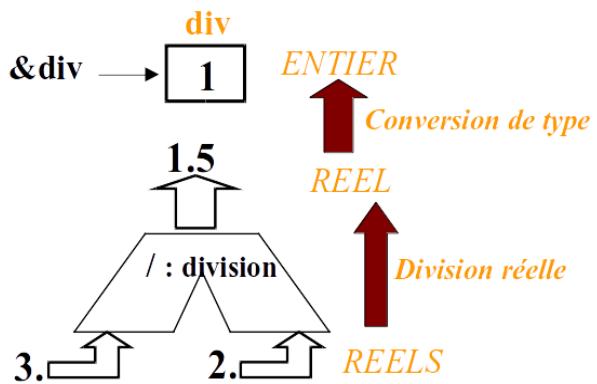


FIGURE 11 : DIVISION REELLE & AFFECTATION DANS ENTIER



### Règle de programmation : compatibilité de types

pour éviter des erreurs de calculs, utiliser, au maximum, des variables de même type dans un calcul arithmétique !!



## FICHE 9 : ALTERNATIVE

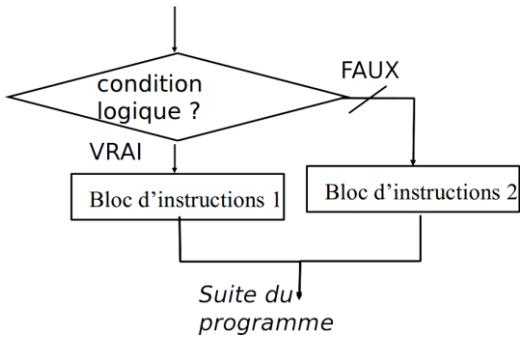




## ALTERNATIVE

Ressources :

[Diapos CHoisir](#)



Objectif :

- ✓ Qu'est-ce qu'une instruction Alternative ?
- ✓ Quels sont les opérateurs logiques
- ✓ Comment écrire un SI-SINON ?
- ✓ Comment écrire un SI-SINONSI ?
- ✓ Comment écrire un CHOIX-CAS ?

## 9.1. QUE SONT LES INSTRUCTIONS ALTERNATIVES ?

Les instructions étudiées jusqu'à présent permettent de construire un programme qui est exécuté séquentiellement du début à la fin du *main()*, avec toujours le même chemin d'exécution possible, voir Figure 12.

Dans ce schéma d'exécution linéaire, un seul chemin existe, comme par exemple dans le programme de calcul de panier suivant :

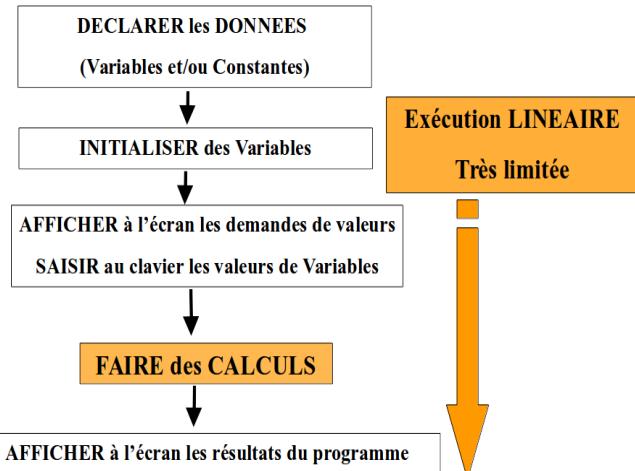


FIGURE 12 : EXECUTION CPU A UN SEUL CHEMIN

```

/********************* CALCUL PRIX D'UN PANIER SANS POSSIBILITE DE REDUCTION *****/
#include <stdio.h>
int main()
{
    unsigned short nbProduit;           // nombre produits achetés
    float          prixTotal;          // prix à payer
    const float    PRIX_UNIT= 10.;     //10 € par produit
    printf("\nNombre de produits achetés : ");
    scanf("%hu",&nbProduit);
    prixTotal= nbProduit * PRIX_UNIT; // calcul du prix à payer
    printf("\nPrix Total : %.2f",prixTotal);
}
  
```

Le problème est d'écrire les instructions de façon à prendre en compte un choix possible :

- si le client n'a pas droit à réduction : le calcul standard est effectué ;
- si le client a droit à réduction : un autre calcul est réalisé.

Les **instructions alternatives** permettent de définir, dans une fonction, **plusieurs chemins d'exécution possibles**, donc des **aiguillages**, voir Figure 13 :

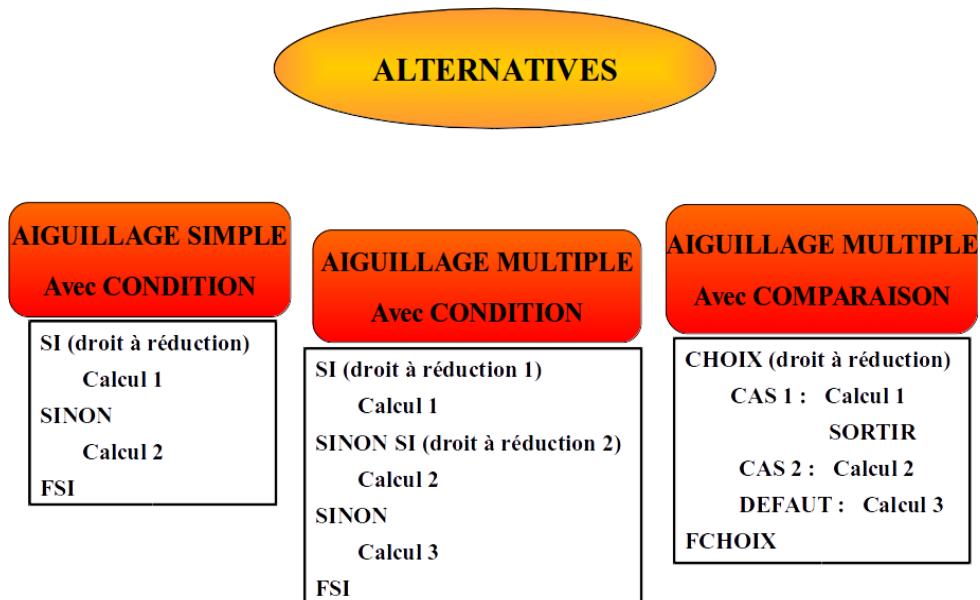


FIGURE 13 : ALTERNATIVES

## 9.2. QUELS SONT LES OPERATEURS LOGIQUES ?

Pour écrire des instructions alternatives, des **conditions logiques** sont requises dans les parenthèses du SI. Ce sont des expressions logiques dont la **valeur est VRAIE ou FAUSSE** : elles servent à faire du raisonnement logique dans le programme.

Dans l'exemple « *SI (a>b) ALORS Afficher("Le maximum est a")* » :

$(a>b)$  est une expression logique que l'UAL évaluera, comme tout calcul avec a valant 3 et b valant 2,  $(a>b)$  est VRAIE.

Une expression logique est constituée :

- d'**opérandes** (a, b, ...) de n'importe quel type simple (int, short, float, char, ...);
- d'**opérateurs logiques**, répartis en deux familles :
  - les **opérateurs logiques de comparaison** (ex. :  $a \neq b$ ,  $a == b$ ,  $c < b$ ), voir Tableau 5 ;
  - les **opérateurs logiques booléens** (ex. :  $\text{NON}(a)$ ,  $b \text{ ET } c$ ), voir Tableau 6.

Opérateur	Nom	En C
$==$	Egal	$==$
$\neq$	Different	$!=$
$>$	Strictement supérieur	$>$
$\geq$	Supérieur ou égal	$\geq$
$<$	Strictement inférieur	$<$
$\leq$	Inférieur ou égal	$\leq$

TABLEAU 5 : OPERATEURS LOGIQUES DE COMPARAISON

Opérateur	Nom	En C															
NON	Non logique	!															
	<table border="1"> <thead> <tr> <th>expression</th> <th>NON(expression)</th> </tr> </thead> <tbody> <tr> <td>VRAI</td> <td>FAUX</td> </tr> <tr> <td>FAUX</td> <td>VRAI</td> </tr> </tbody> </table>	expression	NON(expression)	VRAI	FAUX	FAUX	VRAI										
expression	NON(expression)																
VRAI	FAUX																
FAUX	VRAI																
OU	Ou logique																
	<table border="1"> <thead> <tr> <th>expression1</th> <th>expression2</th> <th>expression1 OU expression2</th> </tr> </thead> <tbody> <tr> <td>FAUX</td> <td>FAUX</td> <td>FAUX</td> </tr> <tr> <td>FAUX</td> <td>VRAI</td> <td>VRAI</td> </tr> <tr> <td>VRAI</td> <td>FAUX</td> <td>VRAI</td> </tr> <tr> <td>VRAI</td> <td>VRAI</td> <td>VRAI</td> </tr> </tbody> </table>	expression1	expression2	expression1 OU expression2	FAUX	FAUX	FAUX	FAUX	VRAI	VRAI	VRAI	FAUX	VRAI	VRAI	VRAI	VRAI	
expression1	expression2	expression1 OU expression2															
FAUX	FAUX	FAUX															
FAUX	VRAI	VRAI															
VRAI	FAUX	VRAI															
VRAI	VRAI	VRAI															
ET	Et logique	&&															
	<table border="1"> <thead> <tr> <th>expression1</th> <th>expression2</th> <th>expression1 ET expression2</th> </tr> </thead> <tbody> <tr> <td>FAUX</td> <td>FAUX</td> <td>FAUX</td> </tr> <tr> <td>FAUX</td> <td>VRAI</td> <td>FAUX</td> </tr> <tr> <td>VRAI</td> <td>FAUX</td> <td>FAUX</td> </tr> <tr> <td>VRAI</td> <td>VRAI</td> <td>VRAI</td> </tr> </tbody> </table>	expression1	expression2	expression1 ET expression2	FAUX	FAUX	FAUX	FAUX	VRAI	FAUX	VRAI	FAUX	FAUX	VRAI	VRAI	VRAI	
expression1	expression2	expression1 ET expression2															
FAUX	FAUX	FAUX															
FAUX	VRAI	FAUX															
VRAI	FAUX	FAUX															
VRAI	VRAI	VRAI															

TABLEAU 6 : OPERATEURS LOGIQUES BOOLEENS

Rappel : l'ordre de priorité de l'ensemble des opérateurs du langage C est rappelé dans Tableau 4 :

Priorité	Opérateur
1	()
2	NON (non logique) — (complément à 1)
3	X (multiplication) / (division) % (modulo)
4	+ (addition) - (soustraction)
5	< > ≤ ≥
6	== (égalité) ≠ (différent)
7	. (et binaire)
8	⊕ (ou exclusif)
9	+ (ou binaire)
10	ET (et logique)
11	OU (ou logique)
12	= (affectation)

TABLEAU 7 : ORDRE DE PRIORITE DES OPERATEURS

Exemple d'évaluation d'une expression logique, r vaut 3.5 (réel) et d vaut 30. (réel)

NON(r==3.5) OU (d>30.)

**ATTENTION à ne pas confondre les opérateurs = (affectation) et == (égalité) !!**

- = permet d'écrire dans la RAM (une valeur est attribuée à une donnée)
- == permet de lire dans la RAM (deux valeurs sont comparées)

$r==3.5$	$NON(r==3.5)$	$d>30.$	
VRAI	FAUX	FAUX	FAUX

## 9.3. INSTRUCTION SI-SINON

### a- Syntaxe et fonctionnement SI-SINON

Le rôle de l'**instruction SI - SINON** est de réaliser un aiguillage à 2 voies exclusives (l'un ou l'autre des chemins est emprunté, mais pas les deux en même temps). Sa simplification, l'instruction **SI** réalise un filtre (par exemple : une protection contre certaines erreurs prévisibles). La syntaxe de l'instruction SI-SINON (traduction en langage C : if else) est la suivante :

```
/*
***** INSTRUCTION SI-SINON *****/
ALGO SI-SINON

DEBUT
    // Aiguillage en fonction d'une condition
    SI (condition logique)
        instructions 1
    SINON // OPTIONNEL
        instructions 2
    FSI
FIN
```

```
/*
***** INSTRUCTION SI-SINON *****/
int main()
{
    // Aiguillage en fonction d'une condition
    if (condition logique)
    {
        instructions 1;
    }
    else // OPTIONNEL
    {
        instructions 2;
    }
}
```

Remarque :

**PAS de ;** à la fin du *if(condition)* et du *else*



#### Règle de programmation : lisibilité du code

- TABULATIONS (indentations) dans les accolades du *if* et du *else*

L'exécution, par le CPU, de l'instruction alternative SI-SINON consiste à choisir entre **instructions 1** et **instructions 2** en fonction de la valeur de la condition logique, voir Erreur ! Source du renvoi introuvable. et Erreur ! Source du renvoi introuvable. :

- Si la condition logique est VRAIE, les instructions 1 du SI/if sont exécutées ;
- Si la condition logique est FAUSSE, le CPU exécute les instructions 2 du SINON/else.

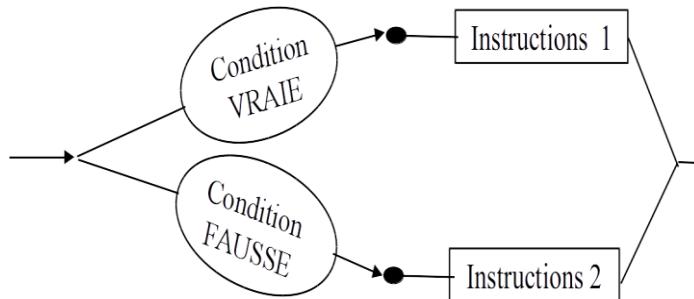


FIGURE 14: FONCTIONNEMENT DU SI-SINON

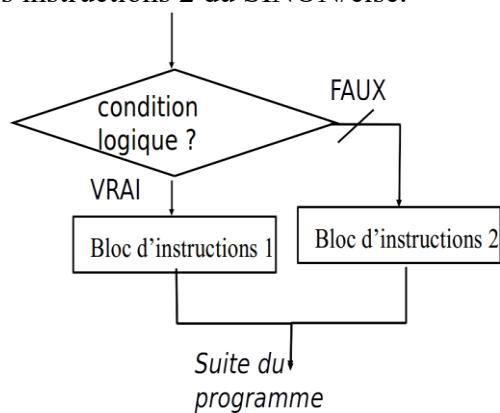


FIGURE 15 : ORGANIGRAMME DU SI-SINON

La simplification de l'instruction SI-SINON en instruction SI est illustrée sur la Figure 16 :

```
*****  
INSTRUCTION SI  
*****  
  
int main()  
{ // Filtre en fonction d'une condition  
    if (condition logique)  
    { instructions 1;  
    }  
}
```

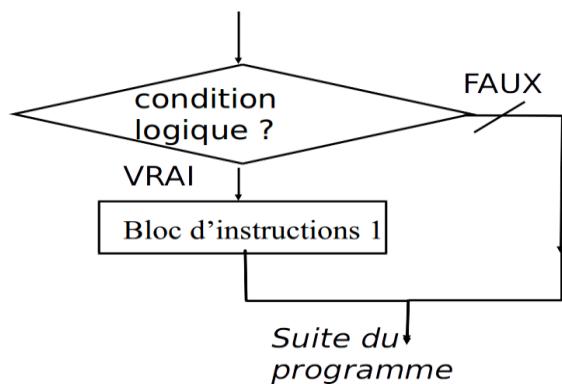


FIGURE 16 : ORGANIGRAMME DU SI

## b- Exemples avec l'instruction SI-SINON

Exemple de calcul d'un panier de courses avec option de réduction : Exactement le même résultat d'exécution de programme peut être obtenu avec différents codes ; il y a souvent plusieurs solutions informatiques à un cahier des charges donné. L'instruction SI-SINON est ici remplacée par l'instruction SI :

```
*****  
ACHATS AVEC POSSIBILITE DE REDUCTION  
*****  
  
#include <stdio.h>  
  
int main()  
{ unsigned short nbProduit;// nb produits achetés  
    float prixTotal; // prix à payer  
    const float PRIX_UNIT= 10.; //10 €/produit  
  
    // saisie du nombre de produits achetés  
    printf("\nNombre de produits achetés : ");  
    scanf("%hu",&nbProduit);  
  
    // calcul prix en fonction du droit à réduction  
    // (20% pour plus de 10 produits)  
    if (nbProduit<=10)  
    { prixTotal= nbProduit * PRIX_UNIT;  
    }  
    else  
    { prixTotal= nbProduit * PRIX_UNIT * 0.8;  
    }  
    printf("\nPrix Total : %.2f",prixTotal);  
}
```

```
*****  
ACHATS AVEC POSSIBILITE DE REDUCTION  
*****  
  
#include <stdio.h>  
  
int main()  
{ unsigned short nbProduit;  
    float prixTotal;  
    const float PRIX_UNIT= 10.;  
  
    // saisie du nombre de produits achetés  
    printf("\nNombre de produits achetés : ");  
    scanf("%hu",&nbProduit);  
  
    // calcul prix en fonction réduction  
    // (20% pour plus de 10 produits)  
    prixTotal= nbProduit * PRIX_UNIT;  
    if (nbProduit>10)  
    { prixTotal= prixTotal * 0.8; }  
    printf("\nPrix Total : %.2f",prixTotal);  
}
```

## Exemple de programmation d'une division avec protection contre les erreurs de calculs prévisibles :

Connaissant les instructions alternatives, il est, maintenant, de votre responsabilité de protéger votre code contre les erreurs envisageables.

```
*****
DIVISION PROTÉGÉE
*****
#include <stdio.h>

int main()
{
    float a, b, div;

    // saisie des 2 réels
    printf("\n\tDonner les 2 reels : ");
    scanf("%f%f",&a,&b);

    // protection division par 0
    if(b != 0.)           // division possible
    {
        div=a/b;
        printf("\n\n\tResultat de la division : %.2f",div);
    }
    else      // message d'erreur : division impossible
    {
        printf("\n\n\tERREUR ! Le diviseur est nul !");
    }
}
```

Les Figure 17 et Figure 18 présentent deux simulations d'exécution du programme de division, dans un cas d'erreur et dans un cas de saisie valide :

Instruction	CPU	Mémoire
1- Déclaration variables	Réservation mémoire	
2- Saisies	Attente, conversion, affectation	
3- if	Evaluation (b≠0): FAUX	
4- else{ Afficher("Erreur") }	Erreur	
5- }	FSI-FIN	

FIGURE 17 : DIVISION AVEC SAISIE INVALIDE

Instruction	CPU	Mémoire
1- Déclaration variables	Réservation mémoire	
2- Saisies	attente, conversion, affectation	
3- if	Evaluation (b≠0): VRAI	
4- { resu= a/b	- calcul: 5./2. - affectation	
5- Afficher(resu)	2.5	

FIGURE 18 : DIVISION AVEC SAISIE VALIDE

## 9.4. INSTRUCTION SI-SINONSI

### a- Syntaxe et fonctionnement SI-SINONSI

Le rôle de l'**instruction SI – SINON SI – SINON** est de réaliser un aiguillage à plusieurs voies exclusives (une seule voie est empruntée). Sa simplification, l'instruction **SI - SINON SI** réalise un aiguillage à plusieurs voies également, mais il est possible de n'utiliser aucune des voies. La syntaxe de l'instruction SI-SINON SI-SINON (traduction en langage C : if else if else) est :

```
*****  
***** INSTRUCTION SI-SINON SI-SINON  
*****  
ALGO SI-SINON SI-SINON  
DEBUT  
  // Aiguillage en fonction de conditions  
  SI (condition 1)  
    instructions 1  
  SINON SI (condition 2)  
    instructions 2  
  SINON // OPTIONNEL  
    instructions 3  
FSI
```

```
*****  
***** INSTRUCTION SI-SINON SI-SINON  
*****  
int main()  
{ // Aiguillage en fonction de conditions  
  if (condition 1)  
  { instructions 1;  
  }  
  else if (condition 2)  
  { instructions 2;  
  }  
  else // OPTIONNEL  
  { instructions 3;  
  }
```

Remarques :

- **PAS de ;** à la fin du *if(condition 1)*, du *else if (condition 2)* et du *else*
- Le nombre de ***else if*** n'est pas limité



#### Règle de programmation : lisibilité du code

- TABULATIONS (indentations) dans les accolades du *if*, des *else if* et du *else*

L'exécution, par le CPU, de l'instruction alternative **SI-SINON SI-SINON** consiste à choisir entre les blocs d'instructions en fonction des valeurs combinées des conditions logiques, voir Figure 19 et Figure 20.

Le CPU évalue successivement les conditions logiques des *if*, de haut en bas :

- Dès qu'une condition est VRAIE, le CPU exécute les instructions associées, puis sort du SI/if.
- Si aucune condition n'est VRAIE, le CPU exécute le dernier SINON/else.

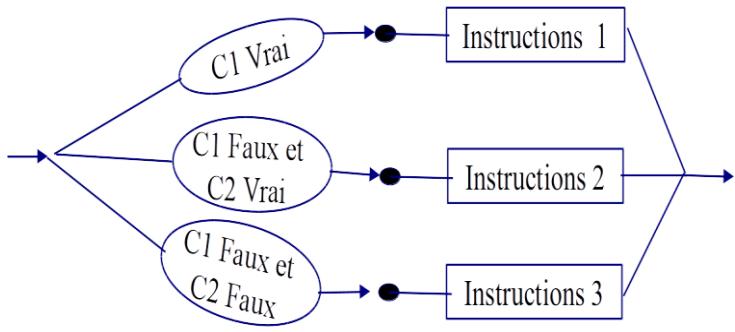


FIGURE 19 : FONCTIONNEMENT DU SI-SINON SI-SINON

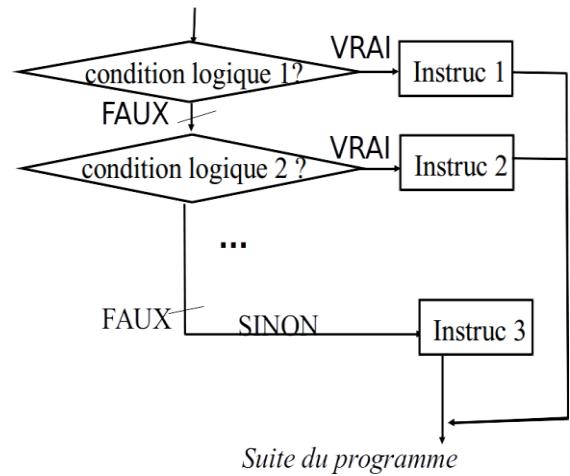


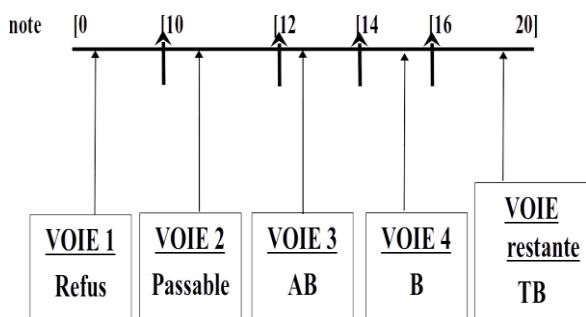
FIGURE 20 : ORGANIGRAMME DU SI-SINON SI-SINON

Le CPU n'arrive sur un *else if* que si les conditions des *if* précédents sont fausses :

	condition 1	condition 2
Exécution instruc 1 si :	V	--
Exécution instruc 2 si :	F	V
Exécution instruc 3 si :	F	F

## b- Exemples avec l'instruction SI-SINONSI

Exemple de programmation d'un découpage d'intervalle : le programme affiche la mention obtenue à un examen :



```

/*
***** MENTION A UN EXAMEN *****
*/
int main()
{
    float note; // NOTE obtenue à l'examen
    printf("\n\tNote : "); scanf("%f",&note);
    // Aiguillage sur la MENTION en fonction note obtenue
    if (note < 10.)
    {
        printf("Refus");
    }
    else if (note < 12.) // note ≥ 10 ET note < 12
    {
        printf("Passable");
    }
    else if (note < 14.) // note ≥ 12 ET note < 14
    {
        printf("AB");
    }
    else if (note < 16.) // note ≥ 14 ET note < 16
    {
        printf("B");
    }
    else // note ≥ 16
    {
        printf("TB");
    }
}
```

## 9.5. INSTRUCTION CHOIX-CAS

### a- Syntaxe et fonctionnement CHOIX-CAS

Le rôle de l'**instruction CHOIX – CAS SORTIR - DEFAUT** est de réaliser un aiguillage à plusieurs voies exclusives (une seule voie est empruntée). La condition d'aiguillage est une simple comparaison. Sa simplification, l'instruction **CHOIX – CAS SORTIR** réalise un aiguillage à plusieurs voies également, mais il est possible de n'utiliser aucune des voies. Pour finir, l'instruction **CHOIX – CAS - DEFAUT** n'est plus un aiguillage : plusieurs CAS peuvent être exécutés séquentiellement. La syntaxe de l'instruction SI-SINON SI-SINON (traduction en langage C : *switch case break default*) est :

```
/*********************************************
INSTRUCTION CHOIX-CAS SORTIR-DEFAUT
*****************************************/
ALGO CHOIX-CAS BREAK-DEFAUT
DEBUT
    // Aiguillage en fonction valeur entière ou caractère
    CHOIX (nomVariable)
        CAS val1 : instructions 1
        SORTIR
        CAS val2 : instructions 2
        SORTIR

        DEFAUT : instructions 3
    FCHOIX
FIN
```

```
/*********************************************
INSTRUCTION CHOIX-CAS SORTIR-DEFAUT
*****************************************/
int main()
{ // Aiguillage en fonction valeur entière ou caractère
    switch (nomVariable)
    {   case val1 : instructions 1;
        break;
        case val2 : instructions 2;
        break;

        default : instructions 3;
    }
}
```

Remarques :

- **PAS de ;** à la fin du *switch*
- *nomVariable* est une variable de type **entier** ou **caractère**
- Le nombre de **CAS n'est pas limité**
- **default** et **break** sont **optionnels**

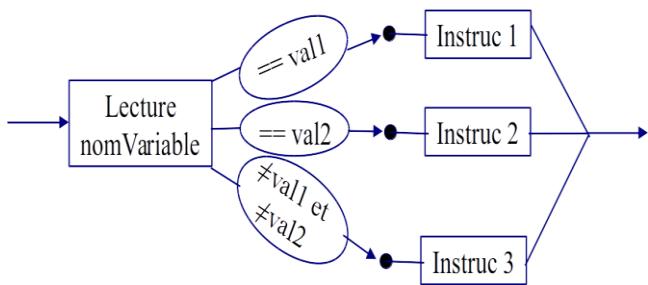


#### Règle de programmation : lisibilité du code

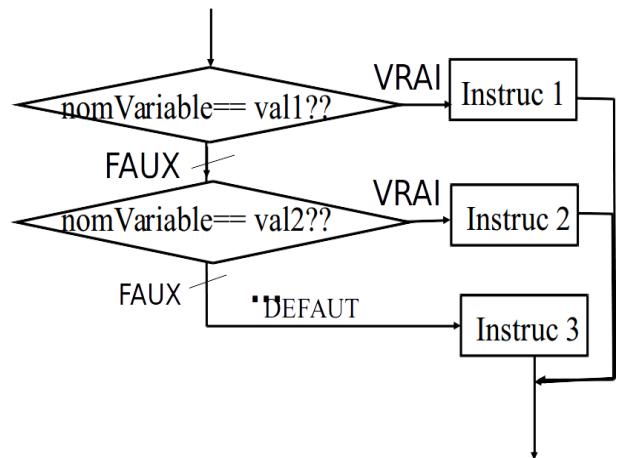
- TABULATIONS (indentations) dans les accolades du *switch* et dans les *case*

L'exécution, par le CPU, de l'instruction alternative **CHOIX-CAS SORTIR-DEFAUT** consiste à choisir entre les blocs d'instructions en fonction de la valeur d'une variable de type **entier** ou **caractère**, voir Figure 19 et Figure 22. Le CPU compare successivement le contenu de *nomVariable* aux valeurs des différents CAS, de haut en bas :

- Dès qu'une égalité est VRAIE, le CPU exécute les instructions associées.
- Arrivé au *break*, le CPU sort de l'instruction CHOIX/*switch*.
- Si aucune égalité n'est VRAIE, le CPU exécute le cas par DEFAUT.



**FIGURE 21 : FONCTIONNEMENT DU CHOIX-CAS SORTIR-DEFAULT**



**FIGURE 22 : ORGANIGRAMME DU CHOIX-CAS SORTIR-DEFAULT**

## b- Exemples avec l'instruction CHOIX-CAS

Exemple de programmation d'une gestion de menu : le programme simule une calculatrice

```
*****
*** CALCULATRICE 2 OPERATIONS ***
*****  

int main()
{ float a, b, resu;
char choixMenu;  

// saisie choix menu
printf("\n\tta- Ad\n\ttb- Sous\n\n\ttChoix: ");
scanf("%c",&choixMenu);
// saisie des opérandes
printf("\n\n\tDonner les 2 reels : ");
scanf("%f%f",&a, &b);
// aiguillage, en fonction du choix utilisateur,
// vers les options du programme
switch(choixMenu)
{ case 'a': resu= a+b; // Addition
break;
case 'b': resu= a-b; // Soustraction
}
printf("\n\n\tResultat de l'operation : %.2f",resu);
}
```

Simulation d'exécution de ce programme :

Instruction	Processeur	Mémoire
1- Déclarations	Réservations mémoire	
2-Saisies	- attente - conversion - affectation	
3- switch,case 'a'	choixMenu=='a' : VRAI	
4- resu=a+b	- calcul: 1.+2. - affectation	
5- break 6- printf()	va à FCHOIX	

# !! ATTENTION, l'oubli de SORTIR change l'exécution du CPU !!

Sans SORTIR/break, le CPU ne sort pas du switch après l'exécution d'un cas, mais exécute les instructions des CAS suivants (comme s'il n'y avait plus d'instruction switch) ; le cas activé est le point d'entrée dans la structure alternative, voir Figure 23 ; ce n'est plus un aiguillage.

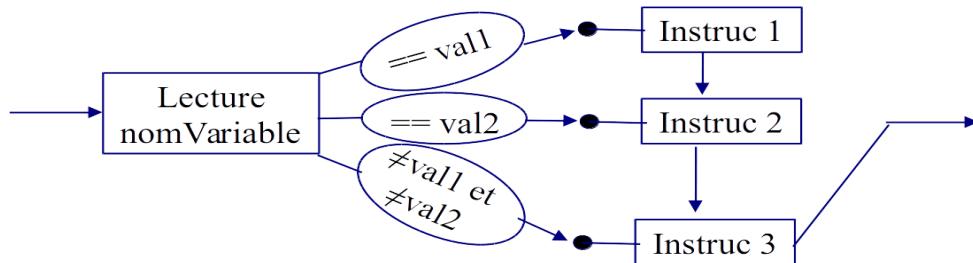


FIGURE 23 : EXECUTION DU CHOIX-CAS SANS BREAK

Exemple de programmation d'une gestion de menu sans SORTIR : la calculatrice

```

/*********************  

CALCULATRICE 2 OPERATIONS sans SORTIR  

*****  

int main()  

{ float a, b, resu;  

  char choixMenu;  

  // saisie choix menu  

  printf("\n\t\ta- Ad\n\t\tb- Sous\n\n\t\tChoix: ");  

  scanf("%c",&choixMenu);  

  // saisie des opérandes  

  printf("\n\n\tDonner les 2 reels : ");  

  scanf("%f%f",&a, &b);  

  // aiguillage, en fonction du choix utilisateur,  

  // vers les options du programme  

  switch(choixMenu)  

  { case 'a': resu= a+b; // Addition  

    // PAS de SORTIR !!  

    case 'b': resu= a-b; // Soustraction  

  }  

  printf("\n\n\tResultat de l'opération : %.2f",resu);
}
  
```

Simulation d'exécution de ce programme :

Instruction	Processeur	Mémoire
1- Déclarations	Réservations mémoire	
2-Saisies	- attente - conversion - affectation choixMenu, a, b	
3- switch,case 'a'	choixMenu=='a' : VRAI	
4- resu= a+b	- calcul: 1.+2. - affectation	
5- resu= a-b	- calcul: 1.-2. - affectation	
6- } 7- printf()	FCHOIX	



## FICHE 10 : FONCTION STANDARD

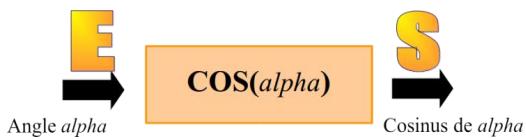




## UTILISER des FONCTIONS STANDARDS

### Ressources :

[Diapos ORGANISER](#)



### Objectif :

- ✓ Qu'est-ce qu'une fonction standard ?
- ✓ Comment l'utiliser ?

### 10.1. DEFINITION

Une **Fonction Standard** est un **bloc d'instructions comprenant ses propres données**. Elle est définie dans une **bibliothèque standard** afin de **réaliser une tâche précise**. Le programmeur n'a donc pas besoin de coder cette tâche et peut utiliser une fonction standard autant qu'il en a besoin dans son programme, à condition d'inclure la bibliothèque standard associée.

Par exemple, la fonction standard `cos()`, qui calcule le cosinus d'un angle *alpha* donné en Radian, fait partie de la bibliothèque standard *math*, voir Figure 24.



FIGURE 24 : FONCTION STANDARD `COS()`

S'agissant de fonctions, trois éléments de vocabulaire doivent être maîtrisés :

1. **PROTOTYPE d'une Fonction** : son **MODE d'EMPLOI / sa DECLARATION**.  
Ainsi, dans le fichier header *math.h* de la bibliothèque standard *math*, le prototype **double cos(double)** indique que la fonction `cos()` :
  - ➔ a un **paramètre en Entrée (E)** *alpha* de type double ; **le programmeur devra transmettre la valeur d'un double dans les parenthèses de la fonction**.
  - ➔ a un **résultat en Sortie (S)** cosinus de *alpha* de type double ; **la fonction renverra à la fin la valeur d'un double** qui pourra être récupérée par la fonction appelant la fonction `cos()`.
2. **DEFINITION d'une Fonction** : son **CODE (ses INSTRUCTIONS et DONNEES)**.  
Ainsi, dans la bibliothèque standard *math* et sous l'en-tête de fonction **double cos(double)** le code de la fonction est écrit entre accolades.
3. **APPEL d'une Fonction** : son **UTILISATION / son EXECUTION**.  
Ainsi, dans une fonction du programmeur -ex. dans le `main()`-, l'instruction d'appel **cosinus = cos(alpha)** permet de demander au CPU d'exécuter la fonction `cos()`, à partir de la valeur double préalablement affectée à la variable *alpha*, et de fournir le résultat par affectation à la variable *cosinus*.

## 10.2. ALGORITHME

L'utilisation d'une fonction standard dans un programme nécessite :

- d'appeler cette fonction -par exemple dans la fonction main()- ;
- d'inclure, en haut du fichier source, le fichier header de sa bibliothèque standard (le header contient la déclaration de la fonction) ;
- la définition de la fonction est « cachée » dans sa bibliothèque standard.

### PROTOTYPE ou DECLARATION DE LA FONCTION STANDARD :

```
/*****************************************************************************  
 HEADER math.h DE LA BIBLIOTHEQUE MATH :  
 CONTIENT LES PROTOTYPES des FONCTIONS MATHEMATIQUES  
******/  
double cos(double)
```

### APPEL DE LA FONCTION STANDARD (UTILISATION-EXECUTION DE CETTE FONCTION) :

```
/*****************************************************************************  
 CALCUL DE COSINUS AVEC LA FONCTION STANDARD  
*****/  
Include bibliothèques standard utilisées  
ALGO main  
    VAR      alpha, cosinus : réels double précision // angle et son cosinus  
DEBUT  
    AFFICHER("Donnez angle (rad) : ")           // saisie angle au clavier  
    SAISIR(&alpha)  
    cosinus= cos(alpha)                      // calcul cosinus de l'angle  
    AFFICHER("Cosinus: valeur",cosinus);  
FIN
```

### DEFINITION DE FONCTION (SES INSTRUCTIONS ET DONNEES) :

```
/*****************************************************************************  
 FONCTION STANDARD cachée dans la BIBLIOTHEQUE STANDARD  
*****/  
ALGO double cos(double alpha)  
    VAR      cosinus : réel double précision // cosinus d'un angle  
DEBUT  
    INSTRUCTIONS      // instructions pour calculer le cosinus de alpha  
    RETOUR(cosinus) // renvoi du cosinus de alpha calculé  
FIN
```

## 10.3. CODAGE EN LANGAGE C

La traduction en C est immédiate, il faut adapter les instructions et utiliser les fonctions standards avec la bibliothèque mathématique :

### PROTOTYPE ou DECLARATION DE LA FONCTION STANDARD :

```
/****************************************************************************
 * HEADER math.h DE LA BIBLIOTHEQUE MATH :
 * CONTIENT LES PROTOTYPES des FONCTIONS MATHEMATIQUES
 */
double cos(double);
```

### APPEL DE LA FONCTION STANDARD (UTILISATION-EXECUTION DE CETTE FONCTION) :

```
/****************************************************************************
 * CALCUL DE COSINUS AVEC LA FONCTION STANDARD
 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main()
{
    double alpha, cosinus;

    printf("\tDonnez angle (rad) : "); // saisie angle au clavier
    scanf("%lf",&alpha);
    cosinus= cos(alpha);           // calcul cosinus de l'angle
    printf("\n\tCosinus: %.2lf\n",cosinus);
}
```

### DEFINITION DE FONCTION (SES INSTRUCTIONS ET DONNEES) :

```
/****************************************************************************
 * FONCTION STANDARD cachée dans la BIBLIOTHEQUE STANDARD
 */
double cos(double alpha)
{
    double cosinus;      // cosinus d'un angle

    INSTRUCTIONS;        // instructions pour calculer le cosinus de alpha
    return(cosinus);     // renvoi du cosinus de alpha calculé
}
```

**Le CPU, exécutant un main() avec appel de fonction, réalise les actions suivantes :**

- Le CPU commence l'exécution au début de la fonction principale main() -c'est le **point d'entrée du programme** : il exécute les instruction une par une, dans l'ordre où elles se présentent (**séquentiellement**), en suivant leur logique.
- Si le CPU rencontre un **appel de fonction** : il met en attente l'exécution de la fonction appelante main().
- L'exécution est redirigée vers la fonction appelée (*si la fonction a un paramètre en Entrée, le CPU se redirige vers la fonction appelée avec une valeur*) ; la **fonction appelée** est exécutée du début à la fin, **séquentiellement**.
- Lorsque l'exécution de la fonction est terminée, le CPU reprend l'exécution de la fonction appelante main() à l'endroit où elle était arrêtée (*si la fonction a un résultat en Sortie, le CPU revient dans le main() avec une valeur*).
- Arrivé à la fin du main(), le CPU sort du programme et décharge le code binaire du programme de la RAM.

Par exemple, la fonction *AfficherAide()*, qui affiche l'aide d'un jeu, est appelée et son exécution est illustrée ci-dessous ; **void** signifie vide, donc pas de donnée échangée (ici *void* dans la parenthèse signifie pas d'Entrée et *void* devant le nom de la fonction signifie pas de Sortie) :

```
*****  
***** HEADER doc.h D'UNE BIBLIOTHEQUE doc :  
***** CONTIENT LE PROTOTYPE de la fonction AfficherAide()  
*****  
void AfficherAide(void);
```

```
*****  
***** AFFICHAGE AIDE D'UN JEU  
*****  
#include <doc.h>  
  
int main()  
{    AfficherAide();  
}
```

```
*****  
***** FONCTION cachée dans une BIBLIOTHEQUE : affichage aide d'un jeu  
*****  
void AfficherAide(void)  
{    printf("\n\t\tAide du jeu de Pendu");  
    printf("Blablabla...");  
}
```

# FICHE 11 : CREER FONCTION

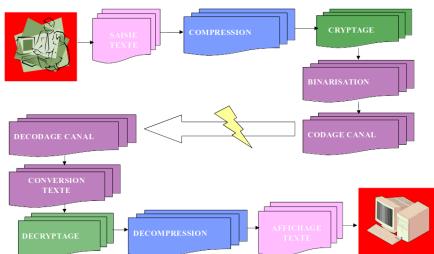




## CREER des FONCTIONS

Ressources :

[Diapos ORGANISER](#)



Objectif :

- ✓ Pourquoi créer ses propres fonctions ?
- ✓ Comment les écrire ?

### 11.1. NECESSITE DES FONCTIONS

Une **Fonction** est un **bloc d'instructions** (module, procédure) comprenant **ses propres données** et réalisant une tâche donnée ; elle peut échanger des informations avec les autres fonctions, voir Figure 25.



FIGURE 25 : BLOC FONCTIONNEL

Faire ses propres fonctions permet de structurer son code, de construire des programmes modulaires composés de plusieurs modules pouvant interagir en échangeant des données. Réaliser un code source structuré présente plusieurs avantages :

- Le programme est découpé logiquement, en fonction des différentes tâches qu'il réalise. Ces tâches sont facilement identifiables par l'instruction d'appel de fonction, qui doit avoir un nom explicite. Le **programme est donc plus lisible lorsqu'il est découpé en fonctions** (pour le programmeur et les collègues amenés à faire évoluer le programme).
- **Lorsqu'un traitement est répétitif** (devant être effectué plusieurs fois dans le programme), il est plus judicieux de ne l'écrire qu'une seule fois (une seule définition de fonction) et de l'appeler autant de fois que nécessaire (plusieurs instructions d'appel de fonction d'une seule ligne). Le **code source est plus court, sa longueur est optimisée**.
- **Le programme modularisé sera plus facile à tester et à valider** : le découpage en fonctions, donc en tâches, permet d'identifier et d'isoler l'ensemble des fonctionnalités du programme à tester.



### Règle de programmation : modularité

- **Un bon programme est modulaire** : il est structuré avec des fonctions (même pour les traitements non répétitifs) !

L'exemple ci-dessous (modularisation d'un programme réalisant une calculatrice élémentaire 4 opérations) montre d'abord le code principal avec des appels de fonction qui est plus lisible, puis le code principal sans fonction qui est trop long car répétitif. Dans l'idéal, la **lecture de la fonction principale permet de comprendre l'architecture de l'ensemble du programme et les principales tâches effectuées**.

```
*****  
CALCULATRICE 4 OPERATIONS : code avec appels de fonction  
*****  
#include <stdio.h>  
  
int main()  
{    int      choixOp ;  
    float      a, b;  
  
    printf("\n\t1 - addit\n\t2-soust\n\t3- multi\n\t4-divis\n");  
    scanf("%d",&choixOp);  
  
    if (choixOp==1)                      // addition  
    {        SaisirValidee(&a,&b);  
            printf("\n\nAddition : %.2f\n",a+b);  
    }  
    else if (choixOp==2)                  // soustraction  
    {        SaisirValidee(&a,&b);  
            printf("\n\nSoustraction : %.2f\n",a-b);  
    }  
    else if (choixOp==3)                  // multiplication  
    {        SaisirValidee(&a,&b);  
            printf("\n\nMultiplication : %.2f\n",a*b);  
    }  
    else if (choixOp==4)                  // division  
    {        SaisirValidee(&a,&b);  
            if (b!=0)      printf("\n\nDivison : %.2f\n",a/b);  
            else          printf("\nErreur");  
    }  
    else  
    {        printf("\nErreur choix menu");  
    }  
}
```

```

/********************* CALCULATRICE 4 OPERATIONS : code sans fonction *****/
#include <stdio.h>

int main()
{
    int      choixOp ;      float    a, b;

    printf("\n\t1- addit\n\t2-soust\n\t3- mult\n\t4-divis\n\t5-Sortir");
    scanf("%d",&choixOp);

    if(choixOp==1)          // addition
    {
        do // Saisie validée des opérandes
        { printf("\n\n Donnez les opérandes : "); scanf("%f%f",&a,&b);
            if (a>50. || b>50.)    printf("\nErreur de saisie");
            } while ((a>50. || b>50.));
        printf("\n\nAddition : %.2f\n",a+b);
    }
    else if(choixOp==2)      // soustraction
    {
        do // Saisie validée des opérandes
        { printf("\n\n Donnez les opérandes : "); scanf("%f%f",&a,&b);
            if (a>50. || b>50.)    printf("\nErreur de saisie");
            } while ((a>50. || b>50.));
        printf("\n\nSoustraction : %.2f\n",a-b);
    }
    else if(choixOp==3)      // multiplication
    {
        do // Saisie validée des opérandes
        { printf("\n\n Donnez les opérandes : "); scanf("%f%f",&a,&b);
            if (a>50. || b>50.)    printf("\nErreur de saisie");
            } while ((a>50. || b>50.));
        printf("\n\nMultiplication : %.2f\n",a*b);
    }
    else if(choixOp==4)      // division
    {
        do // Saisie validée des opérandes
        { printf("\n\n Donnez les opérandes : "); scanf("%f%f",&a,&b);
            if (a>50. || b>50.)    printf("\nErreur de saisie");
            } while ((a>50. || b>50.));
        if (b!=0) printf("\n\nDivison : %.2f\n",a/b);
        else     printf("\nErreur");
    }
    else
    {
        printf("\nErreur choix menu");
    }
}

```

## 11.2. CREATION DE FONCTIONS

L'analyse algorithmique par approche descendante (voir annexe de la Fiche *Programmer*) génère automatiquement les fonctions du code, car elle structure progressivement le programme en différentes tâches.

Un exemple est donné avec la programmation d'un simulateur de transmission numérique entre 2 ordinateurs : Julien et Mathéo s'échangent des e-mails à partir de leur ordinateur, voir Figure 26. Mais que se passe-t-il entre le moment où Julien clique sur envoyer et le moment où Mathéo voit s'afficher le message dans sa boîte mail ?

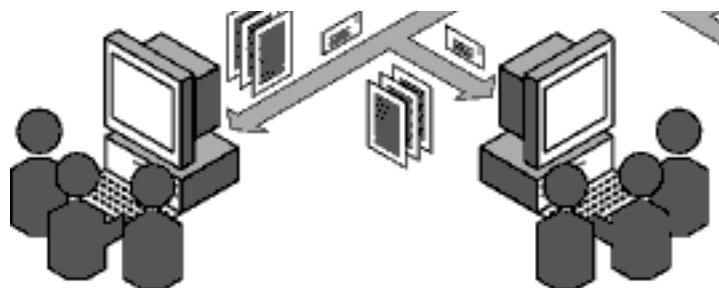


FIGURE 26 : TRANSMISSION NUMERIQUE ENTRE ORDINATEURS

Le programmeur doit identifier les différentes étapes qui permettent de traiter le message textuel afin de le transmettre à un équipement informatique distant. Le découpage du processus technologique produit les différents blocs fonctions de Figure 27.

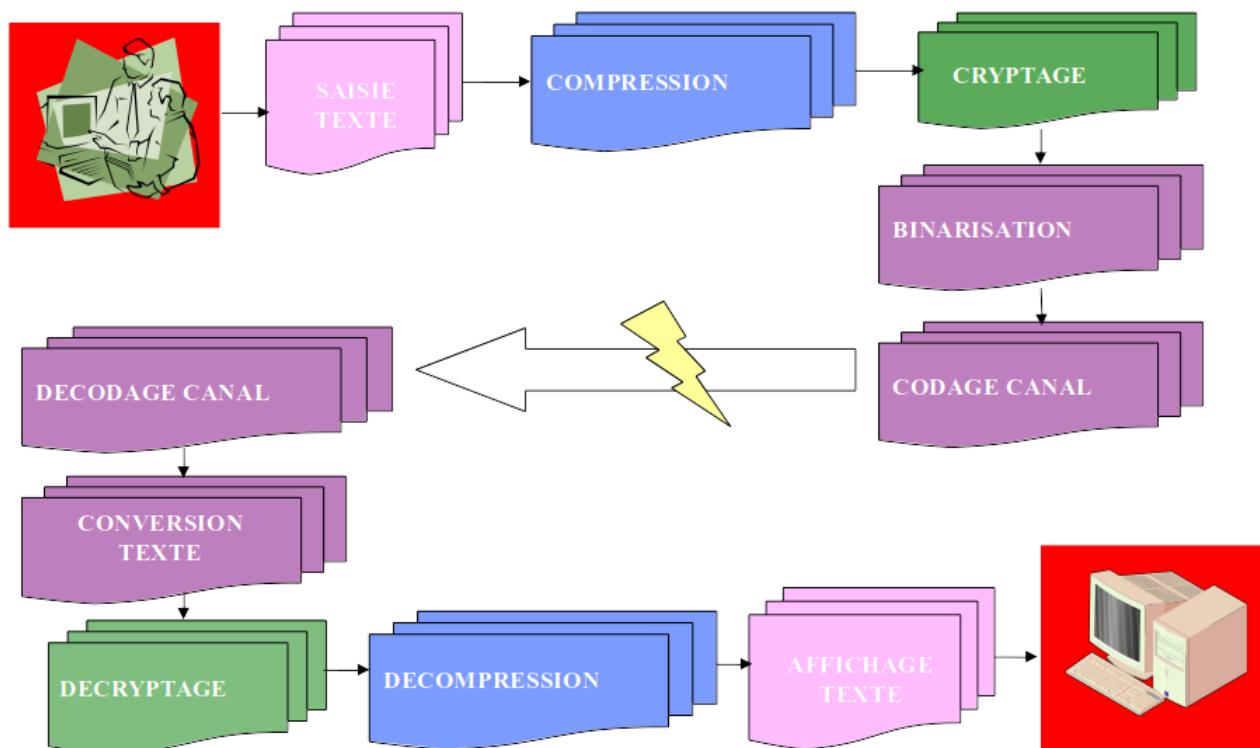


FIGURE 27 : ANALYSE POUR LE SIMULATEUR DE TRANSMISSIONS NUMÉRIQUES

En ce qui concerne une fonction donnée, la Fiche *FonctionStandard* précise comment l'écrire (rappel : 3 parties doivent être présentes, à savoir la **déclaration de fonction ou prototype, l'appel de fonction et la définition de la fonction**). Un algorithme et son code en C sont donnés ci-dessous pour une fonction n'échangeant pas d'informations avec d'autres fonctions (rappel : **void** signifie qu'aucune valeur n'entre, ou ne sort de la fonction) ; le header, nécessaire dans le cas de l'utilisation de fonctions standard, n'est pas considéré ici (le prototype ou déclaration de fonction est alors en haut du fichier source) :

### PROTOTYPE et APPEL DE LA FONCTION :

```
*****
AFFICHAGE AIDE D'UN JEU
*****
void AfficherAide(void)
ALGO main
DEBUT
    AfficherAide()
FIN
```

### DEFINITION DE LA FONCTION :

```
*****
FONCTION d'affichage aide d'un jeu
*****
ALGO void AfficherAide(void)
DEBUT
    AFFICHER("Aide du jeu de Pendu")
    AFFICHER("Blablabla...")
FIN
```

### PROTOTYPE et APPEL DE LA FONCTION :

```
*****
AFFICHAGE AIDE D'UN JEU
*****
void AfficherAide(void);
int main()
{
    AfficherAide();
```

### DEFINITION DE LA FONCTION :

```
*****
FONCTION d'affichage aide d'un jeu
*****
void AfficherAide(void)
{
    printf("\n\t\tAide du jeu de Pendu");
    printf("Blablabla...");
}
```

## 11.3. FONCTIONS ET VARIABLES

Une fonction est un **bloc privé**, qui possède ses instructions et ses propres données ; ces dernières sont appelées **données locales** (elles sont déclarées dans la partie **DONNEES de la fonction**, c'est-à dire en tout début de fonction). Lors de l'exécution du programme une donnée locale à un espace mémoire privé réservé dans la zone RAM de sa fonction. Une donnée locale d'une fonction (donc déclarée dans cette fonction) est privée : elle est visible seulement par les instructions de sa fonction et elle **n'est pas directement accessible depuis une autre fonction**.

Il découle de cette propriété des points importants à prendre en compte dans la manipulation des variables et des fonctions :

- **Des variables locales de différentes fonctions ont donc un espace mémoire distinct, même avec le même nom.** La variable locale *nbEtoile* de la fonction 1 n'a pas le même espace RAM ni la même valeur que la variable locale *nbEtoile* de la fonction 2. Il faut donc faire très attention de ne pas confondre les variables de même nom de fonctions différentes. Une bonne habitude, au début, consiste à **nommer différemment les données de fonctions différentes**.
- **Une fonction ne peut pas utiliser les données locales déclarées dans une autre fonction.** Par exemple, l'espace RAM de la fonction appelée n'est pas directement accessible par la fonction appellante.
- Si plusieurs fonctions ont besoin de travailler avec les mêmes variables ou avec les valeurs de certaines données, le programmeur devra utiliser les **mécanismes d'échanges d'informations entre fonctions : ce sont les paramètres des fonctions et les résultats renvoyés par les fonctions.**

Les notions de portée et de durée de vie des variables sont liées à ces aspects :

1. **PORTEE d'une VARIABLE** : parties du code dans lesquelles une variable est accessible (ces instructions peuvent utiliser cette variable et accéder à sa zone RAM pour y modifier son contenu). **Une règle de base du programmeur consiste à localiser au maximum ses variables dans les fonctions** ; cela permet de suivre plus facilement leur évolution en modes conception ou débogage.
  - ➔ Portée d'une variable locale : la fonction dans laquelle elle est déclarée.
  - ➔ Portée d'une variable globale (une variable globale est déclarée en haut du fichier source, au-dessus de toutes les fonctions) : toutes les fonctions qui sont en-dessous. Les variables globales sont à utiliser avec parcimonie, pour des raisons d'optimisation de l'utilisation de l'espace mémoire, de lisibilité du code source et de facilitation du débogage.
2. **DUREE de VIE d'une VARIABLE** : intervalle de temps pendant lequel la variable existe (elle a un espace RAM dans lequel ses valeurs peuvent être changées).
  - ➔ Durée de vie d'une variable locale : le temps d'exécution de sa fonction. La variable est créée (réservation d'espace RAM) lorsque le CPU entre dans la fonction et elle est détruite (libération de cet espace RAM) lorsque le CPU sort de la fonction.
  - ➔ Durée de vie d'une variable globale : tout le temps d'exécution de l'ensemble du programme -différence temporelle entre l'entrée et la sortie du main() par le CPU-.
  - ➔ Durée de vie d'une variable locale statique (une variable statique est déclarée en début de fonction avec le mot clef *static*) : tout le temps d'exécution du programme. Une variable statique est créée au premier appel d'une fonction, mais n'est pas détruite lorsque le CPU sort de la fonction ; sa zone RAM et sa valeur sont conservées jusqu'à la prochaine exécution de cette fonction. Cette classe de variable permet de mémoriser les valeurs de variable d'un appel à l'autre d'une fonction.

## 11.4. FONCTIONS AVEC PARAMETRES EN ENTREE

Une fonction possède donc ses propres données locales **privées** et n'a pas accès aux données des autres fonctions.

*Alors, comment transmettre la valeur de n à l'intérieur de la fonction *CalculerProduit()* ?*

Les fonctions doivent s'échanger des données : **un paramètre en Entrée permettra, ici, de transmettre une valeur à la fonction au moment de son appel** ; la valeur de n doit être transmise à la fonction par ce mécanisme.

```
*****
*   FONCTION AYANT BESOIN d'une VALEUR pour INITIALISER UNE de ses VARIABLES
*****
ALGO void CalculerProduit( )
  VAR  produit : entier signé long
DEBUT
  produit = 3 x n // élément table multiplication par 3
FIN

*****
FONCTION AYANT BESOIN d'une VALEUR pour INITIALISER UNE de ses VARIABLES
*****
void CalculerProduit( )
{
  int    produit;
  produit = 3 * n; // élément table multiplication par 3
}
```

### a- Syntaxe fonction avec Paramètre en Entrée

Un **PARAMETRE en ENTREE** est déclaré dans les parenthèse de l'en-tête de fonction (au-dessus de la définition de la fonction) ; dans l'exemple ci-dessous le paramètre en E est *n\_f*. Il sert de **variable locale** dans la fonction, mais avec une particularité : la variable paramètre est initialisée, lorsque le CPU entre dans la fonction en mode exécution, **avec la valeur passée dans l'instruction d'appel de cette fonction** ; dans l'exemple ci-dessous le paramètre en E, *n\_f*, est initialisé avec la valeur de *n* -lorsque le CPU redirige son exécution dans la fonction *CalculerProduit()*-.

#### PROTOTYPE et APPEL DE LA FONCTION :

```
*****
APPEL d'une FONCTION avec PARAMETRE en E
*****
void CalculerProduit(int)
ALGO main
  VAR  n : entier signé long
DEBUT
  AFFICHER("Donner n : ")
  SAISIR(&n)
  CalculerProduit(n)
FIN
```

#### DEFINITION DE LA FONCTION :

```
*****
FONCTION avec PARAMETRE en E
*****
ALGO void CalculerProduit(int n_f)
  VAR  produit : entier signé long
DEBUT
  // élément table multiplication par 3
  produit = 3 x n_f
  AFFICHER("Produit : valeur",produit)
FIN
```

## PROTOTYPE et APPEL DE LA FONCTION :

```
*****  
APPEL d'une FONCTION avec PARAMETRE en E  
*****  
void CalculerProduit(int);  
int main()  
{    int      n;  
    printf("Donner n : ");  
    scanf("%d",&n);  
    CalculerProduit(n);  
}
```

## DEFINITION DE LA FONCTION :

```
*****  
FONCTION avec PARAMETRE en E  
*****  
void CalculerProduit(int n_f)  
{    int      produit;  
    // élément table multiplication par 3  
    produit = 3 * n_f;  
    printf("\nProduit : %d",produit);  
}
```

**Un paramètre en E est donc passé par valeur** : lors de l'exécution de l'instruction d'appel de fonction, une valeur de donnée (valeur de *n*) est transmise au paramètre de la fonction (*n\_f*). C'est comme si la variable utilisée pour passer la valeur (*n*) est en lecture seule pour la fonction appelée ; la fonction appelée n'a pas accès à la zone RAM de *n* –qui est une zone mémoire privée de la fonction appelante *main()*- et ne peut donc pas modifier la valeur de la variable *n*. La fonction appelée peut juste recevoir la valeur de *n*, afin d'initialiser la valeur de *n\_f*: la zone RAM de *n\_f* est remplie avec la valeur de *n* ; le paramètre *n\_f* est alors initialisé et peut servir à faire des calculs dans la fonction appelée. Comme les différentes fonctions ont des zones RAM privées, la fonction appelante main() n'a pas accès à la variable n\_f et la fonction appelée CalculerProduit() n'a pas accès à la variable n. Il y a juste transmission de valeur de la fonction appelante vers la fonction appelée.

Remarque : il est judicieux, au début, de nommer différemment les variables locales de fonctions différentes, afin de ne pas mélanger ses variables. Cela concerne, en particulier, les paramètres formels - dans la définition *n\_f* - et les paramètres effectifs -dans l'appel *n*- des fonctions.

## b- Exécution fonction avec Paramètre en Entrée

Illustration de l'exécution, par le CPU, d'un appel de fonction avec paramètre en E pour le code suivant :

```
*****  
APPEL d'une FONCTION avec PARAMETRE en E  
*****  
void CalculerProduit(int n_f);  
int main()  
{    int      n;  
    printf("Donner n : ");  
    scanf("%d",&n);  
    CalculerProduit(n);  
}
```

```
*****  
FONCTION avec PARAMETRE en E  
*****  
void CalculerProduit(int n_f)  
{    int      produit;  
    // élément table multiplication par 3  
    produit = 3 * n_f;  
    printf("\nProduit : %d",produit);  
}
```

Les étapes d'exécution sont :

1. Exécution de la fonction principale *main()* :

- ♦ Déclaration et saisie de n : un espace RAM est réservé pour n dans la zone mémoire du *main()* et n reçoit 4 du clavier.
- ♦ **Appel fonction avec passage valeur 4 : *CalculerProduit(4)*.**

2. Exécution de la fonction appelée *CalculerProduit()* :

- ♦ **Initialisation paramètre en E :** Le CPU initialise *n\_f* avec la valeur de n (*n\_f = n = 4*) dans la zone RAM de la fonction *CalculerProduit()* (Figure 28 montre l'initialisation effectuée par le CPU lors de l'appel de fonction).
- ♦ Calcul du produit et affichage du résultat (12).
- ♦ Retour à la fonction *main()*.

3. Fin de l'exécution du *main()*.



FIGURE 28 : INITIALISATION D'UN PARAMETRE EN ENTREE

Remarque : il est possible de déclarer plusieurs paramètres en Entrée, même avec des types différents ; ils doivent être séparés par une virgule dans l'en-tête de définition de fonction. Dans l'instruction d'appel, l'ordre des valeurs doit correspondre à l'ordre des paramètres formels.

## 11.5. FONCTIONS AVEC RESULTAT EN SORTIE

Une fonction possède ses propres données locales **privées** et n'a pas accès aux données des autres fonctions.

*Alors, comment transmettre la valeur de produit, en retour, à la fonction *main()* ?*

Les fonctions doivent s'échanger des données : **un résultat en Sortie permettra, ici, de renvoyer une valeur à la fonction appelante *main()* au moment du retour de la fonction appelée** ; la valeur de *produit* doit être transmise au *main()* par ce mécanisme.

```
/*
***** FONCTION AYANT BESOIN d'une VALEUR pour INITIALISER UNE de ses VARIABLES *****/
ALGO void CalculerProduit( )
  VAR  produit : entier signé long
  DEBUT
    produit = 3 x n // élément table multiplication par 3
  FIN
/*
***** FONCTION AYANT BESOIN d'une VALEUR pour INITIALISER UNE de ses VARIABLES *****/
void CalculerProduit( )
{
  int  produit;
  produit = 3 * n; // élément table multiplication par 3
}
```

## a- Syntaxe fonction avec Résultat en Sortie

Un **RESULTAT en SORTIE** est la valeur d'une variable locale de la fonction appelée qui est renvoyée à la fonction appelante (avec l'instruction *return*), afin d'être récupérée dans une variable locale de la fonction appelante ; dans l'exemple ci-dessous le résultat en S est *prod\_f*, il sera récupéré dans *prod*.

### PROTOTYPE et APPEL DE LA FONCTION :

```
*****
APPEL d'une FONCTION avec RESULTAT en S
*****  
int CalculerProduit(int)  
ALGO main  
    VAR n, prod : entier signé long  
DEBUT  
    AFFICHER("Donner n : ")  
    SAISIR(&n)  
    prod= CalculerProduit(n)  
    AFFICHER("Produit : valeur",prod)  
FIN
```

### DEFINITION DE LA FONCTION :

```
*****
FONCTION avec RESULTAT en S
*****  
ALGO int CalculerProduit(int n_f)  
    VAR prod_f : entier signé long  
DEBUT  
    // élément table multiplication par 3  
    prod_f = 3 x n_f  
    RETOUR(prod_f)  
FIN
```

```
*****
APPEL d'une FONCTION avec RESULTAT en S
*****  
int CalculerProduit(int);  
int main()  
{    int n, prod;  
    printf("Donner n : ");  
    scanf("%d",&n);  
    prod= CalculerProduit(n);  
    printf("\nProduit : %d",prod);  
}
```

```
*****
FONCTION avec RESULTAT en S
*****  
int CalculerProduit(int n_f)  
{    int prod_f;  
    // élément table multiplication par 3  
    prod_f = 3 * n_f;  
    return(prod_f);  
}
```

**Un résultat en S est donc passé par valeur** : lorsque le CPU exécute l'instruction *return*, il redirige son exécution vers la fonction *main()*, en transportant une valeur de donnée (valeur de *prod\_f*), qui est affectée à la variable locale du *main()* *prod*. Comme les différentes fonctions ont des zones RAM privées, la fonction appelante *main()* n'a pas accès à la variable *prod\_f* et la fonction appelée *CalculerProduit()* n'a pas accès à la variable *prod*. Il y a juste transmission de valeur de la fonction appelée vers la fonction appelante.

Remarque : il est judicieux, au début, de **nommer différemment les variables locales de fonctions différentes**, afin de ne pas mélanger ses variables. Cela concerne, en particulier, **les variables résultats dans la définition et dans l'appel des fonctions**.

## b- Exécution fonction avec Résultat en Sortie

Illustration de l'exécution, par le CPU, d'un appel de fonction avec résultat en S pour le code suivant :

```
/*********************************************
 * APPEL d'une FONCTION avec RESULTAT en S
 *****/
int CalculerProduit(int);
int main()
{
    int n, prod;
    printf("Donner n : "); scanf("%d",&n);
    prod= CalculerProduit(n);
    printf("\nProduit : %d",prod);
}
```

```
/*********************************************
 * FONCTION avec RESULTAT en S
 *****/
int CalculerProduit(int n_f)
{
    int prod_f;
    // élément table multiplication par 3
    prod_f = 3 * n_f;
    return(prod_f);
}
```

Les étapes d'exécution sont :

1. Exécution de la fonction principale *main()* :
  - Déclaration et saisie de n : un espace RAM est réservé pour n dans la zone mémoire du *main()* et n reçoit 4 du clavier.
  - **Appel fonction avec passage valeur 4 : *CalculerProduit(4)*.**
2. Exécution de la fonction appelée *CalculerProduit()* :
  - Initialisation paramètre en E : Le CPU initialise n\_f avec la valeur de n ( $n_f = n = 4$ ) dans la zone RAM de la fonction *CalculerProduit()*.
  - Calcul du produit (12) et stockage dans la variable locale *prod\_f*.
  - **Retour au *main()* avec renvoi de la valeur de produit : 12.**
3. Fin de l'exécution du *main()*.
  - **Récupération du résultat dans la variable locale prod : *prod= prod\_f = 12*** (Figure 29 montre le retour de valeur effectué par le CPU lors du retour de fonction).
  - Affichage du résultat.

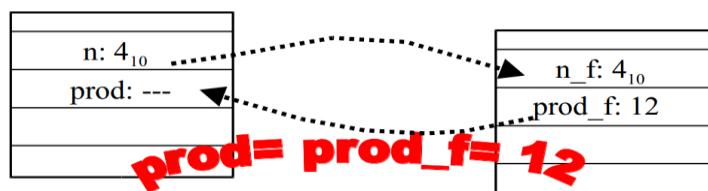


FIGURE 29 : RECUPERATION D'UN RESULTAT EN SORTIE

Remarques :

- Il est impossible de renvoyer plusieurs résultats en S : 1 fonction peut renvoyer 1 seule valeur.
- L'instruction *return* provoque, automatiquement, la sortie de la fonction appelée ; toute instruction sous *return* sera ignorée par le CPU.



## FICHE 12 : BOUCLE

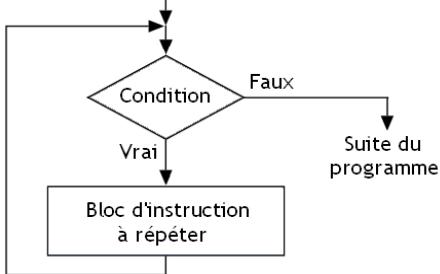




## BOUCLE

### Ressources :

[Diapos RECOMMENCER](#)



### Objectif :

- ✓ Qu'est-ce qu'une instruction itérative (Boucle) ?
- ✓ Comment écrire une boucle POUR ?
- ✓ Comment écrire les boucles TANT QUE ?

## 12.1. QUE SONT LES INSTRUCTIONS ITERATIVES ?

$$\begin{aligned}0 \times 2 &= 0 \\1 \times 2 &= 2 \\2 \times 2 &= 4 \\&\vdots \\10 \times 2 &= 20\end{aligned}$$

Soit un programme qui affiche la table de multiplication par 2, selon ce format :

```
*****  
TABLE MULTIPLICATION  
*****  
int main()  
{ int produit;  
  // multiplication par 0  
  produit= 0*2;  
  printf("\n0 x 2 = %d",produit);  
  // multiplication par 1  
  produit= 1*2;  
  printf("\n1 x 2 = %d",produit);  
  ... 11 fois ... !!  
}
```

Il serait judicieux d'éviter de réécrire des instructions (voir ci-contre) et de les présenter de façon plus synthétique, en les paramétrant et en indiquant que le CPU doit les recommencer :

*Instructions à faire 11 fois, avec i qui commence à 0 et qui finit à 10 :*

```
produit= i*2;  
printf("%dx2=%d",i,produit);
```

## 12.2. INSTRUCTION POUR

### a- Syntaxe et fonctionnement POUR

Le rôle de l'**instruction POUR** est de répéter l'exécution d'un bloc d'instructions un nombre connu de fois : c'est une boucle avec compteur. La syntaxe de l'instruction POUR (traduction en langage C : for) est :

```
*****  
***** INSTRUCTION POUR *****  
*****  
ALGO POUR  
DEBUT  
    // Répétition avec comptage croissant  
    POUR i de valInit à valFin, pas  
        instructions  
FPOUR  
FIN
```

```
*****  
***** INSTRUCTION POUR *****  
*****  
int main()  
{    int i;  
    // Répétition avec comptage croissant  
    for ( i=valInit ; i<=valFin ; i=i+pas )  
    {        instructions;  
    }  

```

Remarque :

- **PAS de ;** à la fin du *for*
- Variable compteur (variable de boucle) *i* est obligatoirement de type **entier** ou **caractère**



#### Règle de programmation : lisibilité du code

- TABULATIONS (indentations) dans les accolades du *for*

Questions de vocabulaire :

**Initialisation : *i=valInit***

*Affectation de la valeur de début de comptage*

```
for ( i=valInit ; i<=valFin ; i=i+pas )  
    instructions;  
}
```

**Incrémantation : *i=i+pas***

*Augmentation de la valeur du compteur à chaque reprise de boucle*

**Condition de poursuite de boucle : *i<=valFin* (cas croissant)**

*Condition qui détermine la poursuite de la boucle : si cette condition est VRAIE, la boucle continue*  
*Condition de fin = NON(Condition de poursuite) : si cette condition est VRAIE, la boucle s'arrête.*

Précisions sur le PAS, incrémentation du compteur entre 2 itérations :

- Si le pas est positif : le compteur *i* compte (sens croissant) ;
- Si le pas est négatif : le compteur *i* décompte (sens décroissant) ; dans ce cas, la condition de poursuite est inversée (*i>=valFin*).

L'exécution, par le CPU, de l'instruction itérative POUR consiste à recommencer l'exécution des instructions de la boucle tant que la variable compteur ne dépasse pas la valeur finale, voir Figure 30 et Figure 31 :

- Le compteur i démarre, la première fois, à vallInit ; **puis le CPU vérifie la condition de poursuite** :  $i \leq valFin$  (pour un comptage croissant ici) ;
- Si la condition de poursuite est VRAIE, le CPU exécute la boucle, puis revient au POUR ;
- Lorsque le CPU revient au POUR : i est incrémenté de pas ; puis le CPU vérifie la condition de poursuite de boucle :  $i \leq valFin$  ;
- Quand la condition de poursuite devient FAUSSE, le CPU sort du POUR/for.

INSTRUCTION	PROCESSEUR
1- <b>for</b>	→ a- initialisation : $i = valeurInit$ b- test poursuite de boucle $i \leq valFin$ : <b>VRAI</b>
2- {	→ exécution des instructions
3- } : <b>for</b>	→ (automatique) a- incrémentation : $i = i + pas$ b- test poursuite boucle $i \leq valFin$ : <b>VRAI</b>
4- {	→ exécution des instructions
5- } : <b>for</b>	→ (automatique) a- incrémentation : $i = i + pas$ b- test poursuite boucle $i \leq valFin$ : <b>FAUX</b>
6- }	→ suite du programme après la fin du POUR

FIGURE 30 : TABLEAU D'EXECUTION DE LA BOUCLE POUR

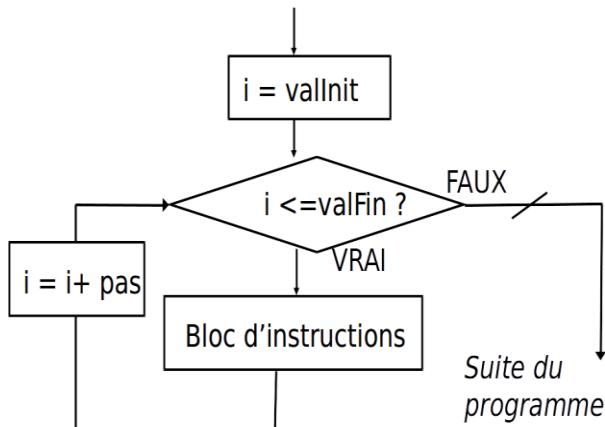


FIGURE 31 : ORGANIGRAMME DE LA BOUCLE POUR

Remarque : lorsque le programmeur écrit et met au point une boucle POUR, il doit bien vérifier qu'elle fonctionne pour les valeurs limites (vallInit et valFin).

## b- Exemples avec l'instruction POUR

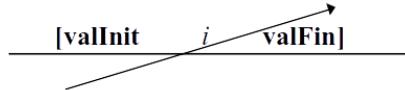
Exemple de calcul cumulatif grâce à une boucle POUR : une variable est initialisée avant la boucle, puis elle est augmentée d'une certaine quantité à chaque fois qu'une itération de la boucle s'exécute (la structure itérative permet de cumuler des quantités dans une même variable). La variable est mise à jour à chaque passage dans la boucle : la valeur précédente est écrasée par la nouvelle valeur calculée :

```
/****************************************************************************
 * CUMUL DE 4 ENTIERS SAISIS AU CLAVIER
 */
int main()
{   int a, somme, i;
    somme= 0; // initialisation de la somme, alors 1e cumul sera: 0+1eentier

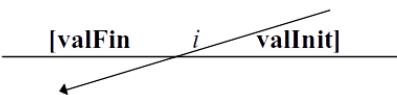
    // comptage de 4 saisies
    for (i=1 ; i<=4 ; i++)
    {   // saisie d'un entier
        printf("Donner un entier : ");
        scanf("%d",&a);
        // à chaque passage: cumul de a+valeur précédente de somme
        somme= somme + a;
    }
    // affichage de la somme des 4 entiers
    printf("somme : %d",somme);
}
```

Exemple de boucle POUR utilisée pour faire du décomptage : suivant le signe du pas, il faut faire attention à la condition de poursuite :

 Comptage : pas > 0  
• valInit ≤ valFin (croissant)



 Décomptage : pas < 0  
• valInit ≥ valFin (décroissant)



 Condition poursuite de Boucle :

$$i \leq \text{valFin}$$

 Condition fin de Boucle :

$$i > \text{valFin}$$

**for(i=1;i<=10;i=i+2)**

$$i = \{1, 3, 5, 7, 9\}$$

$$i \geq \text{valFin}$$

 Condition fin de Boucle :

$$i < \text{valFin}$$

**for(i=50;i>=47;i=i-1)**

$$i = \{50, 49, 48, 47\}$$

## 12.3. INSTRUCTIONS TANT QUE

### a- Syntaxe et fonctionnement TANT QUE

Le rôle des **instructions FAIRE TANT QUE** et **TANT QUE** est de répéter l'exécution d'un bloc d'instructions tant qu'une condition est vraie : c'est une boucle conditionnelle. La syntaxe des instructions FAIRE TANT QUE et TANT QUE (traduction en langage C : *do while* et *while*) est :

```
*****  
INSTRUCTION FAIRE TANT QUE  
*****  
ALGO FAIRE TANT QUE  
  
DEBUT  
    // Répétition sur condition  
    FAIRE  
        instructions  
    TANT QUE (condition)  
FIN
```

```
*****  
INSTRUCTION FAIRE TANT QUE  
*****  
int main()  
{    // Répétition sur condition  
    do  
    {        instructions;  
    } while (condition);  
}
```

```
*****  
INSTRUCTION TANT QUE  
*****  
ALGO TANT QUE  
  
DEBUT  
    // Répétition sur condition  
    TANT QUE (condition)  
        instructions  
    FTANTQUE  
FIN
```

```
*****  
INSTRUCTION TANT QUE  
*****  
int main()  
{    // Répétition sur condition  
    while (condition)  
    {        instructions;  
    }  
}
```

Remarque :

- **UN ; à la fin du while() seulement pour le *do while* !!!**



#### Règle de programmation : lisibilité du code

- TABULATIONS (indentations) dans les accolades du *while* et du *do while*

**L'exécution CPU des instructions itératives FAIRE TANT QUE et TANT QUE consiste à recommencer l'exécution des instructions de la boucle tant que la condition de poursuite de boucle est VRAIE ; dès que la condition de fin de boucle devient VRAIE, la boucle s'arrête.**

En ce qui concerne l'exécution CPU de la boucle FAIRE TANT QUE, voir Figure 32 et Figure 33 :

- Le CPU passe le DO et exécute les instructions de boucle ;
- Le CPU évalue la condition de poursuite de boucle ;
- Si la condition est VRAIE, le CPU réexécute la boucle ;
- Lorsque le CPU arrive au WHILE, le CPU évalue de nouveau la condition ;
- Quand la condition devient FAUSSE, le CPU sort du TANTQUE/do while.

En ce qui concerne l'exécution CPU de la boucle TANT QUE, voir Figure 34 et Figure 35 :

- Le CPU évalue la condition de poursuite de boucle ;
- Si la condition est VRAIE, le CPU exécute la boucle, puis revient au WHILE ;
- De retour au WHILE, le CPU évalue de nouveau la condition ;
- Quand la condition devient FAUSSE, le CPU sort du TANTQUE/while.

La différence entre les deux exécutions est que le FAIRE TANT QUE s'exécute toujours au moins une fois, puisque la condition de poursuite est évaluée en fin de boucle (contrairement au TANT QUE, dans lequel il est possible que le CPU ne passe pas du tout).

INSTRUCTION	PROCESSEUR
1- <b>do</b> _____	exécution des instructions de la boucle
3- <b>do</b> _____	exécution des instructions de la boucle
5- <b>do</b> _____	exécution des instructions de la boucle
6- <b>while</b> _____	évaluation condition logique de poursuite : FAUX
7- _____	suite du programme après la fin du TANTQUE

FIGURE 32 : TALEAU D'EXECUTION DU FAIRE-TANT QUE

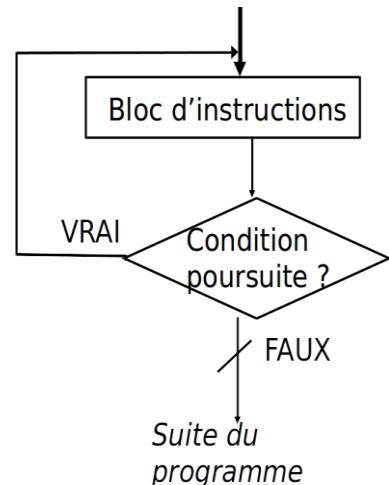


FIGURE 33 : ORGANIGRAMME DU FAIRE-TANT QUE

INSTRUCTION	PROCESSEUR
2- {	exécution des instructions de la boucle
4- {	exécution des instructions de la boucle
5- } : while	évaluation condition logique de poursuite : FAUX
6-	suite du programme après la fin du TANT QUE

FIGURE 34 : TABLEAU D'EXECUTION DU TANT QUE

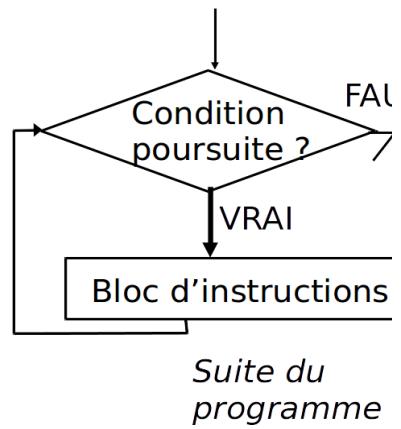


FIGURE 35 : ORGANIGRAMME DU TANT QUE

Remarque : lorsque le programmeur écrit et met au point une des boucles TANT QUE, il doit bien vérifier qu'elle fonctionne bien pour les conditions limites.

## b- Exemples avec l'instruction TANT QUE

Exemple de transformation d'une boucle POUR en une boucle TANT QUE : une boucle POUR peut toujours être écrite sous la forme d'une boucle tant que : l'initialisation est avant la boucle, l'incrémentation dans la boucle et la condition de poursuite dans les parenthèses du TANT QUE. La boucle TANT QUE est la boucle la plus généraliste, sa condition logique peut être complexe. Le choix de la boucle POUR est possible lorsque le programmeur sait exactement combien de fois sa boucle doit tourner.

```
/*
***** INSTRUCTION POUR *****
***** */

int main()
{
    int i;

    for ( i=valInit ; i<=valFin ; i=i+pas )
    {
        instructions;
    }
}
```

```
/*
***** INSTRUCTION FAIRE TANT QUE *****
***** */

int main()
{
    int i;

    i=valInit;
    do
    {
        instructions;
        i=i+pas
    } while (i<=valFin);
}
```

Exemple de reprise de programme avec une boucle FAIRE TANT QUE : le programme réalise une calculatrice avec reprise si l'utilisateur veut recommencer le programme. La boucle de reprise du programme permet d'éviter d'avoir à relancer le programme (RUN) pour une nouvelle exécution.

```
/*
***** CALCULATRICE AVEC REPRISE DE PROGRAMME *****/
int main()
{
    float a, b, resu ; int choixMenu;

    do // reprise du programme tant que l'utilisateur ne veut pas sortir
    { printf("1) addition,\n2) soustraction\n3) SORTIR\n\tChoix :");
        scanf(" %d",&choixMenu);
        printf("Donner 2 réels:"); scanf("%f%f",&a,&b);

        if (choixMenu==1) // addition
        { resu= a+b; }
        if (choixMenu==2) // soustraction
        { resu= a-b; }
        printf("Resultat de l'opération : %.2f",resu);

    } while (choixMenu!=3);
}
```

Exemple de saisie validée de valeurs au clavier avec une boucle FAIRE TANT QUE : le programme saisit une valeur, puis vérifie sa validité, en cas d'erreur de l'utilisateur, il l'en informe, puis donne une seconde chance à l'utilisateur en recommençant la saisie. Le programmeur doit vérifier la validité de toutes les saisies du programme et demander une nouvelle valeur en cas d'erreur. Il doit également bien informer l'utilisateur de ce qui doit être tapé au clavier et des erreurs qu'il commet.

```
/*
***** LE PROGRAMME SAISIT UN ENTIER ∈ [5,150] *****/
int main()
{
    int x;

    do // reprise de saisie si saisie invalide
    { printf("Donner une valeur comprise entre 5 et 150 : ");
        scanf("%d",&x);

        if (x<5 || x>150 ) // message d'erreur si saisie invalide
        { printf("Erreur de saisie"); }

    } while (x<5 || x>150 );
}
```

## FICHE 13 : TABLEAU

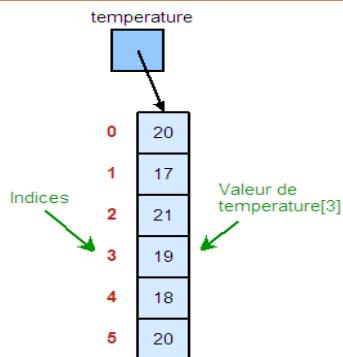




## TYPE TABLEAU

Ressources :

[Diapos MEMORISER](#)



Objectif :

- ✓ Qu'est-ce qu'un tableau ?
- ✓ Comment le manipuler ?

### 13.1. DEFINITION

Le **type tableau** est un **type de données** qui caractérise une **suite de cases mémoires permettant de stocker des données de même type simple** (float, int, ...). Un tableau peut avoir plusieurs dimensions :

- Un **tableau à 1 dimension** correspond à un vecteur : les **cases** sont alignées en une ligne ou en une colonne ; elles **sont identifiées par un indice** (qui **commence à la valeur 0**).
- Un tableau à 2 dimensions est une matrice : les cases ont un numéro de ligne et un numéro de colonne.

La Figure 36 suivante montre une variable de type tableau dans la RAM et sous forme de représentation simplifiée.

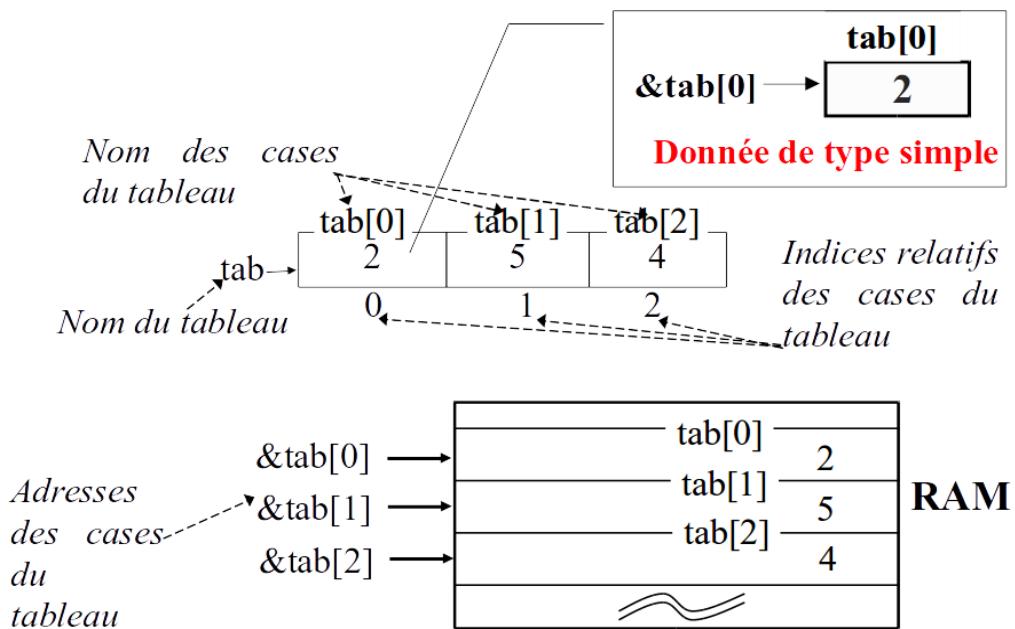
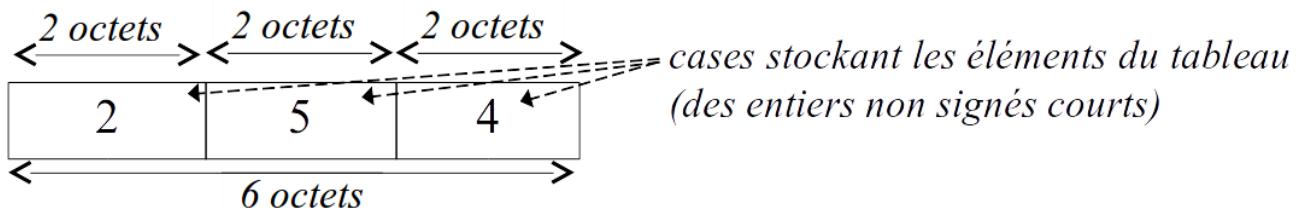


FIGURE 36 : REPRESENTATION D'UN TABLEAU

Le tableau se nomme *tab*. Chacune des cases du tableau est identifiée par un indice, tel que la première case est numérotée 0, la deuxième 1, ..., et la dernière case a pour indice (NBCASE-1). NBCASE, nombre de cases du tableau, est souvent déclaré comme une constante dans le code. **Une case du tableau se nomme à partir du nom du tableau et de son indice *i* entre crochet : *tab[i]*.** *tab[i]* correspond, tout simplement, une variable de type simple (un float, int, char, ...), caractérisée par un nom *tab[i]*, une adresse *&tab[i]* et un contenu de case. La représentation du tableau dans la RAM montre les cases avec leur adresse.

Exemple, le tableau suivant possède 3 cases contenant des unsigned short :



Remarque : le **nom du tableau *tab*** désigne aussi l'**adresse du début du tableau** (l'adresse de la première case du tableau), autrement dit ***tab=&tab[0]***. On dit que *tab* est un pointeur : il pointe sur la case RAM où est stockée la première case du tableau, voir Figure 37.

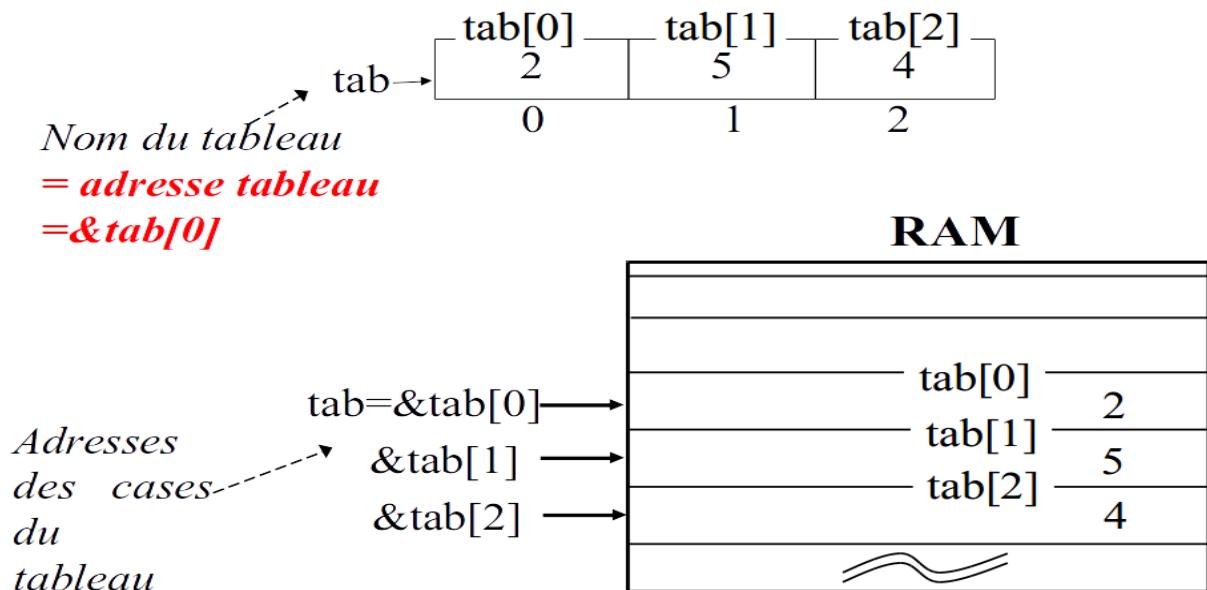


FIGURE 37 : POINTEUR SUR UN TABLEAU

## 13.2. ALGORITHME ET C (DECLARATION VARIABLE TABLEAU)

L'instruction de déclaration de variables de type tableau à 1 dimension permet de n'avoir à déclarer qu'une seule variable, au lieu de 100 par exemple, pour stocker 100 valeurs de même type. L'utilisation d'une constante pour définir le nombre de case du tableau est une bonne pratique. La déclaration du tableau s'écrit dans la *partie données* d'une fonction :

```
/*****************************************************************************  
 DECLARATION de VARIABLE de TYPE TABLEAU à 1 DIMENSION  
*****/  
ALGO DeclarerVariableTableau // PARTIE DONNEES  
    // définition du nombre de cases du tableau en constante : BONNE PRATIQUE  
    CONST NBCASE= 50 : entier non signé court  
    VAR nomVarTableau[NBCASE] : TYPEcase // TYPEcase : type simple  
        tabMesures[100] : entier non signe court          // tableau de 100 short  
        tabNotes[NBCASE] : réel double précision         // tableau de 50 double  
DEBUT  
FIN
```

```
/*****************************************************************************  
 DECLARATION de VARIABLE de TYPE TABLEAU à 1 DIMENSION  
*****/  
int main()  
{ // PARTIE DONNEES  
    // définition du nombre de cases du tableau en constante : BONNE PRATIQUE  
    const unsigned short NBCASE= 50;  
    TYPEcase nomVarTableau[NBCASE]; // TYPEcase : type simple  
    short tabMesures[100];           // tableau de 100 short  
    double tabNotes[NBCASE];        // tableau de 50 double  
  
    // PARTIE INSTRUCTIONS  
}
```

### Règle de programmation : lisibilité du code (nommage des variables Tableau)



- Noms explicites, pas trop longs ;
- 1<sup>o</sup> lettre en minuscule (variable locale) ;
- Préciser rôle variable avec un commentaire explicatif.

L'exécution, par le CPU, de l'instruction de déclaration de variable tableau de NBCASE case entraîne la réservation d'un espace mémoire vide dans la RAM, tel que :

- son adresse est *nomTableau* ou *&nomTableau[0]*;
- le nom de ses cases est : *nomTableau[0]*, *nomTableau[1]*, ..., *nomTableau[NBCASE-1]*;
- la place mémoire réservée est égale à : NBCASE x longueur du type simple choisi;
- le contenu des cases est vide (ou indéterminé).

### 13.3. INITIALISATION D'UNE VARIABLE TABLEAU

Après avoir déclaré une variable de type tableau, il sera nécessaire de l'**initialiser**, c'est-à-dire de donner un contenu à ses cases. Plusieurs modes d'initialisation sont possibles :

1. **Initialisation dans la déclaration** : cette solution n'est possible que pour des tableaux pas trop grands dont les valeurs sont connues à-priori ; le nombre de valeurs doit correspondre au nombre de cases du tableau et les valeurs doivent être écrites de la première à la dernière case :

```
*****  
INITIALISATION de VARIABLE de TYPE  
TABLEAU dans la DECLARATION  
*****  
ALGO InitTabDeclar // PARTIE DONNEES  
CONST NBCASE= 50 : entier non signé court  
VAR  
nomTab[NBCASE]= {val1, val2, ...} : TYPEcase  
// 10 mesures en Ohms  
tabMesures[10]= {-2, 6, 100, -50, 0, 0, 1, 8, 45, -42} : esc  
// 5 notes du module 1  
tabNotes[NBCASE]= {10.5, 20., 5., 14.5, 18.} : rdp  
DEBUT  
FIN
```

```
*****  
INITIALISATION dans la DECLARATION  
*****  
int main()  
{ // PARTIE DONNEES  
const unsigned short NBCASE= 5;  
TYPEcase nomTab[NBCASE]= {val1, val2, ...};  
// 10 mesures en Ohms  
short tabMesures[10]= {-2,6,100,-50,0,0,1,8,45,-42};  
// 5 notes du module 1  
double tabNotes[NBCASE]= {10.5,20.,5.,14.5,18.};  
// PARTIE INSTRUCTIONS  
}
```

tabMesures → 

-2	6	100	-50	0	0	1	8	45	-42
----	---	-----	-----	---	---	---	---	----	-----

 tabNotes → 

10.5	20.	5.	14.5	18.
------	-----	----	------	-----

2. **Initialisation par affectations** : cette solution nécessite des affectations case à case automatisées dans une boucle POUR (ce qui implique que les valeurs stockées dans les cases sont, soit identiques, soit suivent une relation mathématique connue). **La boucle POUR permet de parcourir les cases une par une** -ici de la première d'indice 0 à la dernière d'indice (NBCASE-1)- : **à chaque fois que la boucle POUR recommence**, l'indice i de parcours du tableau est incrémenté de 1 et le CPU se place sur la case suivante, afin d'y effectuer un traitement.

```
*****  
INITIALISATION par AFFECTATIONS  
*****  
ALGO InitTabAffect // PARTIE DONNEES  
CONST NBCASE= 10 : ensc  
VAR // 10 mesures en Ohms  
tabMesures[NBCASE], i : esc  
DEBUT // PARTIE INSTRUCTIONS  
POUR i de 0 à (NBCASE-1), pas de 1  
tabMesures[i]= 0  
FPOUR  
FIN
```

```
*****  
INITIALISATION par AFFECTATIONS  
*****  
int main()  
{ // PARTIE DONNEES  
const unsigned short NBCASE= 10;  
short tabMesures[NBCASE], i;  
  
// PARTIE INSTRUCTIONS  
for (i=0 ; i<NBCASE ; i++)  
{ tabMesures[i]= 0; }
```

3. **Initialisation par saisies clavier** : cette solution n'est possible que pour des tableaux pas trop grands dont les valeurs doivent être fournies par l'utilisateur du programme sur la clavier; le nombre de valeurs doit correspondre au nombre de cases du tableau et le programme doit bien indiquer à l'utilisateur dans quel ordre il doit donner les valeurs des cases :

```
*****
INITIALISATION de VARIABLE de TYPE
TABLEAU par SAISIES CLAVIER
*****
ALGO InitTabAffect // PARTIE DONNEES
CONST NBCASE= 10 : ensc
VAR      // 10 mesures en Ohms
tabMesures[NBCASE], i : esc
DEBUT    // PARTIE INSTRUCTIONS
POUR i de 0 à (NBCASE-1), pas de 1
    AFFICHER("Case n° valeur : ",i)
    SAISIR(&tabMesures[i])
FPOUR
FIN
```

```
*****
INITIALISATION de VARIABLE de TYPE
TABLEAU par SAISIES CLAVIER
*****
int main()
{ // PARTIE DONNEES
    const unsigned short NBCASE= 10;
    // 10 mesures en Ohms
    short tabMesures[NBCASE], i;

    // PARTIE INSTRUCTIONS
    for (i=0 ; i<NBCASE ; i++)
    {
        printf ("\nCase n° %hd : ",i);
        scanf("%hd",&tabMesures[i]);
    }
}
```

4. **Initialisation par générations de nombres aléatoires** : cette solution permet de mettre des valeurs au hasard dans les cases du tableau. Le programme doit d'abord initialiser le générateur de nombres aléatoires (une seule fois en début de programme), afin que les valeurs tirées au hasard diffèrent d'une exécution à l'autre du programme. Ensuite, le programmeur doit définir le domaine dans lequel les valeurs doivent être choisies. Pour choisir un intervalle entre 0 et une valeur maximale, il suffit de paramétrier avec le maximum de l'intervalle ; pour avoir un minimum autre que 0, il faut effectuer un décalage (offset) de l'intervalle de tirage.

```
*****
INITIALISATION de VARIABLE de TYPE
TABLEAU par GÉNÉRATION ALÉATOIRE
*****
ALGO InitTabAffect // PARTIE DONNEES
CONST NBCASE= 10 : ensc
VAR      // 10 mesures en Ohms
tabMesures[NBCASE], i : esc
DEBUT    // PARTIE INSTRUCTIONS
// tirages aléatoires dans [0,15]
POUR i de 0 à (NBCASE-1), pas de 1
    tabMesures[i]= Aleatoire(0,15)
FPOUR
FIN
```

```
*****
INITIALISATION de VARIABLE de TYPE
TABLEAU par GÉNÉRATION ALÉATOIRE
*****
#include <stdlib.h>
#include <time.h>
int main()
{ const unsigned short NBCASE= 10;
short tabMesures[NBCASE], i;
// tirages aléatoires dans [0,15]
rand(time(NULL));
for (i=0 ; i<NBCASE ; i++)
{
    tabMesures[i]= rand()%16;
}
}
```

## 13.4. UTILISER UNE VARIABLE TABLEAU

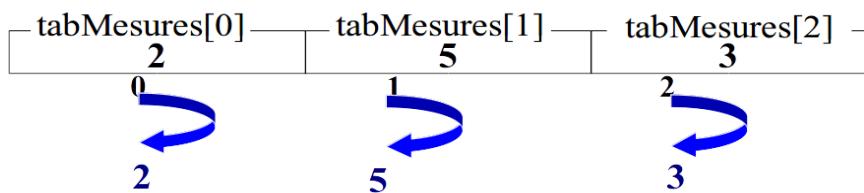
Une fois déclaré et initialisé, un tableau peut être utilisé pour diverses opérations de lecture ou d'écriture.

### a- Affichage d'un tableau

Pour afficher le contenu des cases d'un tableau à 1 dimension, en langage C, il faut parcourir les cases une par une afin de lire et de transmettre, successivement, leur contenu sur l'écran. Une case se manipule comme une variable de type simple.

```
/****************************************************************************
 *          AFFICHAGE d'un TABLEAU sur l'ECRAN
 */
ALGO AfficherTableau
CONST NBCASE= 2 : ensc
VAR tabMesures[NBCASE]= {1,2}, i : esc
DEBUT
POUR i de 0 à (NBCASE-1), pas de 1
    AFFICHER("Case n°valeur1 : valeur2",i,tabMesures[i])
FPOUR
FIN
```

```
/****************************************************************************
 *          AFFICHAGE d'un TABLEAU sur l'ECRAN
 */
int main()
{
    const unsigned short NBCASE= 2;
    short tabMesures[NBCASE]= {1,2}, i;
    for (i=0 ; i<NBCASE ; i++)
    {
        printf ("\nCase n°%hd : %hd",i,tabMesures[i]);
    }
}
```



## b- Stockage de valeurs dans un tableau

Les cases d'un tableau peuvent être parcourues afin de recevoir des résultats de calculs. L'exemple ci-dessous permet de stocker la table de multiplication par 2 dans un tableau :

```
/*
***** STOCKAGE TABLE de MULTIPLICATION par 2 *****
***** */

ALGO StockerTableau
CONST NBCASE= 5 : ensc

VAR tabProduit[NBCASE], i : esl

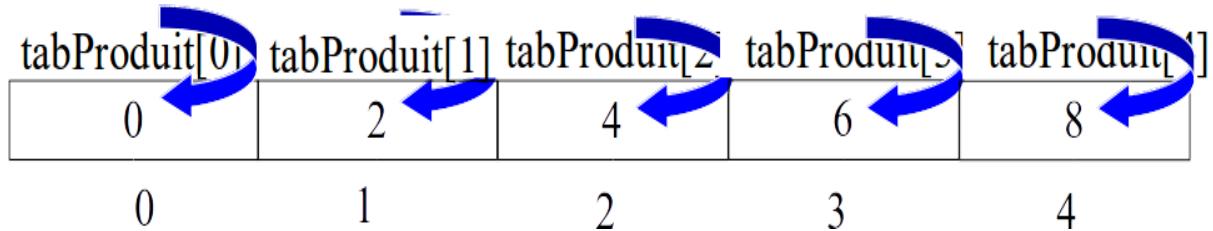
DEBUT
POUR i de 0 à (NBCASE-1), pas de 1
    tabProduit[i]= ix2
    AFFICHER("valeur1 x 2 = valeur2\n",i,tabProduit[i])
FPOUR
FIN
```

```
/*
***** STOCKAGE TABLE de MULTIPLICATION par 2 *****
***** */

int main()
{ const unsigned short NBCASE= 5;

    int tabProduit[NBCASE], i;

    for (i=0 ; i<NBCASE ; i++)
    {   tabProduit[i]= i*2;
        printf ("%d x 2 = %d\n",i,tabProduit[i]);
    }
}
```

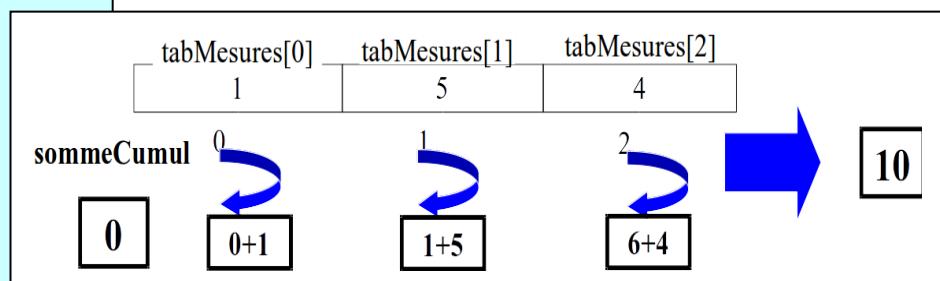


## c- Calculer avec des valeurs stockées dans un tableau

Les **cases d'un tableau peuvent être parcourues afin d'être lues pour servir à faire des calculs**. Les deux exemples ci-dessous permettent de calculer la somme et le produit cumulés des valeurs contenues dans un tableau (addition et produits de toutes les cases). Le principe du cumul consiste à utiliser une variable de cumul. Cette variable est initialisée à l'élément neutre de l'opération de cumul (0 pour l'addition), afin que la valeur d'initialisation n'interfère pas avec les valeurs du tableau. **Cumuler le contenu des cases du tableau dans la variable de cumul implique de parcourir les cases, une par une, et de rajouter le contenu de chaque case au contenu précédent de la variable de cumul.**

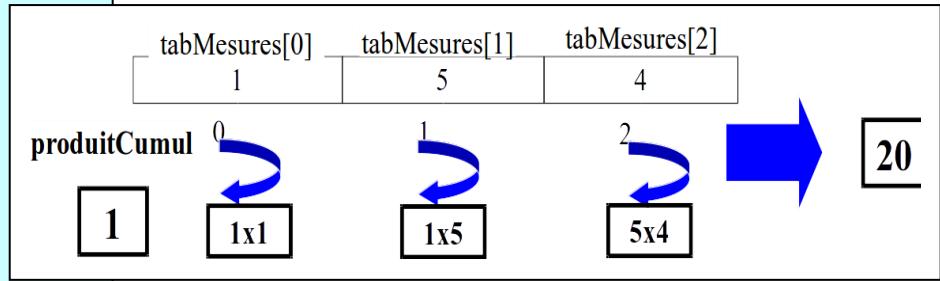
```
/*
***** SOMME CUMULEE d'un TABLEAU *****
*****/
```

```
int main()
{ const unsigned short NBCASE= 3;
  int tabMes[NBCASE]={1,5,4}, i ;
  int somCumul;
  somCumul= 0;
  for (i=0 ; i<NBCASE ; i++)
  {
    somCumul= somCumul+tabMes[i];
  }
  printf ("\nSomme mesures : %d",somCumul);
}
```



```
/*
***** PRODUIT CUMULE d'un TABLEAU *****
*****/
```

```
int main()
{ const unsigned short NBCASE= 3;
  int tabMes[NBCASE]={1,5,4}, i ;
  int prodCumul;
  prodCumul= 1;
  for (i=0 ; i<NBCASE ; i++)
  {
    prodCumul= prodCumul * tabMes[i];
  }
  printf ("\nProduit mesures : %d",prodCumul);
}
```



## FICHE 14 : STRUCTURE

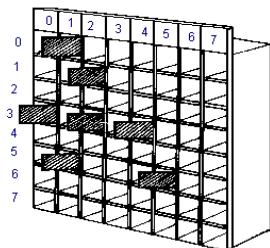




## TYPE STRUCTURE

Ressources :

[Diapos MEMORISER](#)



Objectif :

- ✓ Qu'est-ce qu'une structure ?
- ✓ Comment l'utiliser ?

### 14.1. DEFINITION

Le **type structure** est un type de donnée complexe qui permet de lier dans une même entité différents champs de types éventuellement différents (types simples, tableaux ou pointeurs). Contrairement aux tableaux qui nécessitent le même type pour toutes ses cases, une structure est un assemblage de variables qui peuvent avoir différents types :



Un type structuré est créé par le programmeur selon ses besoins de regroupement d'informations au sein d'une entité globale. Ce type permet de déclarer, ensuite, des variables structurées contenant différents champs les décrivant. Il permet de modéliser la réalité de façon plus intuitive : une structure structCrayon, par exemple, contiendrait des champs modélisant les caractéristiques physiques du crayon (couleur, taille, type de mine, ...) ; les variables crayon\_Hugo, crayon\_Lise de ce type structuré seraient alors définies avec les valeurs les décrivant (le crayon\_Hugo est rouge, HB et mesure 8cm). Ainsi, pour décrire toutes les propriétés d'un crayon, une seule variable est nécessaire.

Pour manipuler une structure, le programmeur doit :

- créer le type structure, en définissant ses champs (type et nom de variable associés à chaque propriété de la structure) ;
- créer une variable, à partir de ce type structuré, et affecter des valeurs à ses champs, afin de la personnaliser.

### 14.2. ALGORITHME ET C (DECLARATION TYPE STRUCTURE)

L'instruction de déclaration d'un type structuré permet de définir un modèle d'entité, avec ses propriétés sous forme de champs (type et nom de variable). La déclaration de structure s'écrit en haut du fichier source, au-dessus de la fonction `main()`. En langage C, le ; à la fin définition du type structuré est obligatoire :

```
*****
DECLARATION d'un TYPE STRUCTURE
*****/
STRUCT structNomTypeStructure

    nomChamps1 : TYPEchamps1
    nomChamps2 : TYPEchamps2
    ...
FSTRUCT

ALGO DeclarerTypeStructure

DEBUT
FIN
```

```
*****
DECLARATION d'un TYPE STRUCTURE
*****/
struct structNomTypeStructure

{   TYPEchamps1 nomChamps1;
    TYPEchamps2 nomChamps2;
    ...
};

int main()
{}
```



#### Règle de programmation : lisibilité du code (nommage des types structure)

- Noms explicites, pas trop longs ;
- préfixe "struct", afin de différencier type et variable structurés
- Préciser rôle du type avec un commentaire explicatif.

Par exemple, la structure suivante décrit une personne :

```
*****
TYPE STRUCTURE décrivant une personne
*****/
struct structPersonne
{
    char    initialNom;
    char    initialPrenom;
    int     age;
    double  salaire;
};

int main()
{}
```

## 14.3. ALGORITHME ET C (DECLARATION VARIABLE STRUCTURE)

L'instruction de déclaration d'une variable de type structuré permet de préciser les données strcuturées qui pourront être utilisées dans le programme. Elle s'écrit dans la *partie données* de la fonction qui utilise cette variable :

```
/*
 * DECLARATION VARIABLE STRUCTURE
 */
STRUCT structTypeStructure
{
    nomChamps1 : TYPEchamps1
    nomChamps2 : TYPEchamps2
    ...
} FSTRUCT

ALGO DeclarerVarStructure
VAR nomVar : structTypeStructure
DEBUT
FIN
```

```
/*
 * DECLARATION VARIABLE STRUCTURE
 */
struct structTypeStructure
{
    TYPEchamps1 nomChamps1;
    TYPEchamps2 nomChamps2;
    ...
};

int main()
{
    struct structTypeStructure nomVar;
}
```



### Règle de programmation : lisibilité du code (nommage des variables de type structure)

- Noms explicites, pas trop longs ;
- 1<sup>o</sup> lettre en minuscule (variable locale) ;
- Préciser rôle variable avec un commentaire explicatif.

Par exemple, la variable florian est une personne avec les caractéristiques de la structure *structPersonne* :

```
/*
 * STRUCTURE décrivant une personne
 */
struct structPersonne
{
    char initialNom;
    char initialPrenom;
    int age;
    double salaire;
};
int main()
{
    struct structPersonne florian;
```

## 14.4. ALGORITHME ET C (UTILISATION VARIABLE STRUCTURE)

Une fois le type structuré déclaré, puis la variable structurée déclarée, il est nécessaire d'initialiser cette dernière, afin de préciser les valeurs de ses propriétés. Pour accéder à un champ d'une variable structurée, il faut utiliser l'opérateur de champs (point) entre le nom de la variable et le nom du champ. Plusieurs modes d'initialisation sont possibles :

1. **Initialisation dans la déclaration** : cette solution ressemble à l'initialisation des tableaux dans la déclaration. Le nombre de valeurs doit correspondre au nombre de champs du type structuré associé et les valeurs doivent être écrites dans l'ordre des champs :

```
/*********************  
 * STRUCTURE décrivant un POINT  
 * *******************/  
struct structPoint  
{    double    x;      // abscisse  
     double    y;      // ordonnée  
};  
int main()  
{    struct structPoint point= {0,0};  
}
```

2. **Initialisation par affectations** : cette solution permet d'attribuer des valeurs aux champs d'une variable structurée dans les parties instructions des fonctions (il est aussi possible de faire une affectation globale de la forme : pointA= pointB) :

```
/*********************  
 * STRUCTURE décrivant un POINT  
 * *******************/  
struct structPoint  
{    double    x;      // abscisse  
     double    y;      // ordonnée  
};  
int main()  
{    struct structPoint pointA;  
        pointA.x= 0;  
        pointA.y= 0;  
}
```

```
/*********************  
 * STRUCTURE décrivant une personne  
 * *******************/  
struct structPersonne  
{    char      initialNom;  
    char      initialPrenom;  
    int       age;  
    double    salaire;  
};  
int main()  
{    struct structPersonne florian;  
        florian.initialNom= 'T';  
        florian.initialPrenom= 'F';  
        florian.age= 18;  
        florian.salaire= 1900.;  
}
```

## FICHE 15 : POINTEUR





## TYPE POINTEUR

Ressources :

[Diapos MEMORISER](#)

Objectif :

ptReel —→ (NULL)

- ✓ Qu'est-ce qu'un pointeur ?
- ✓ Comment le manipuler ?

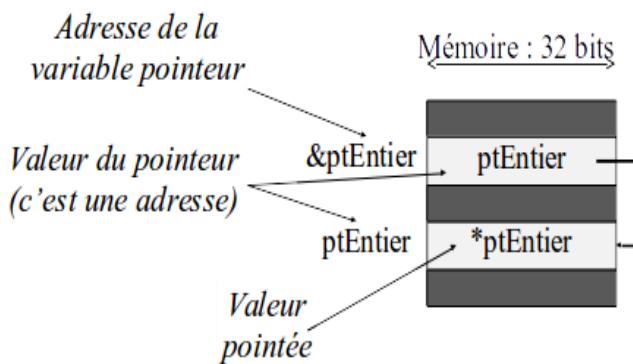
### 15.1. DEFINITION

Le **type pointeur** est un type de donnée. Une **variable de type pointeur** a une valeur (contenu de sa case RAM) qui est l'adresse mémoire d'une autre **variable de type simple** (int, short, float, double, char...).

Une **variable de type pointeur** pointerà donc sur une autre case RAM et pourra accéder au contenu mémoire pointé, afin de le modifier (le pointeur permet de « pirater » une variable de type simple, c'est-à-dire d'accéder à son contenu « à l'insu » de la variable pointée).

Un type pointeur définit donc sur quel type simple la variable pointeur va pointer. Il est question de pointeur sur int ou de pointeur sur float, ... Le **signe qui caractérise un type pointeur en langage C est le signe \***. Ce signe étoile permet de préciser un type pointeur et permet aussi de qualifier le contenu RAM pointé par une variable pointeur. Remarque : &a (adresse de la variable a de type simple) est un pointeur sur a et, si a est un int, alors &a est un pointeur sur int (ce qui s'écrit int \*).

La Figure 38 représente une variable de type pointeur sur int dans la RAM, nommé *ptEntier*. La variable *ptEntier* a donc un espace réservé pour stocker son contenu (comme toute variable) : elle dispose d'un espace RAM étiqueté *ptEntier* et se trouvant à l'adresse *&ptEntier*. Or, comme le contenu mémoire d'un pointeur (sa valeur!) est une adresse RAM, la case RAM du pointeur *ptEntier* contient une adresse et pointe donc sur une autre case RAM, dont l'adresse est *ptEntier*. Cette case pointée (ici recevant des valeurs de type int) se nomme *\*ptEntier* ; son contenu sera modifiable via le pointeur *ptEntier*.



Un pointeur permet donc d'accéder à une case RAM, afin d'y faire des modifications ; cette case pointée peut appartenir à une autre variable du programme (par exemple *nbPoints*). Le pointeur *ptEntier* pourra, alors, agir comme un pirate qui ira modifier la valeur de la variable piratée *nbPoints*.

FIGURE 38 : VARIABLE DE TYPE POINTEUR

La représentation simplifiée de **Figure 39** permet de dessiner des pointeurs en se concentrant sur l'essentiel : la zone pointée dont la valeur sera modifiée par le pointeur. Le **nom de pointeur apparaît comme l'adresse de la zone pointée**, selon le schéma de représentation des variables de type simple.

ptEntier → \*ptEntier

FIGURE 39 : REPRESENTATION SIMPLIFIEE D'UNE VARIABLE POINTEUR

## 15.2. ALGORITHME ET C (DECLARATION VARIABLE POINTEUR)

L'**instruction de déclaration d'une variable de type pointeur** permet d'indiquer les pointeurs que le programme manipulera. Elle s'écrit dans la *partie données* de la fonction qui utilise cette variable :

```
/*
***** DECLARATION de VARIABLE de TYPE POINTEUR *****/
ALGO DeclarerVariablePointeur // PARTIE DONNEES
    VAR  *nomPointeur : typeVarPointée // type simple numérique ou caractère
        *ptReel : réel simple précision      // pointeur sur un float
        *ptEntier : entier signé long      // pointeur sur un int
DEBUT
FIN

/*
***** DECLARATION de VARIABLE de TYPE POINTEUR *****/
int main()
{   typeVarPointée *nomPointeur; // type pointé simple numérique ou caractère
    float          *ptReel;       // pointeur sur un float
    int            *ptEntier;     // pointeur sur un int
}
```

### Règle de programmation : lisibilité du code (nommage des variables pointeur)

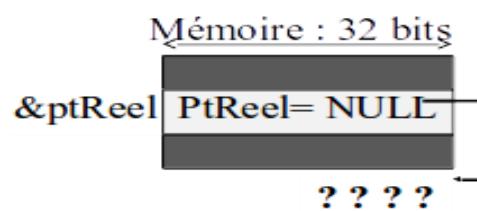


- Noms explicites, pas trop longs ;
- 1<sup>o</sup> lettre en minuscule (variable locale) ;
- Préciser rôle variable avec un commentaire explicatif.

L'**exécution, par le CPU, de l'instruction de déclaration de variable pointeur entraîne la réservation d'un espace mémoire vide**, tel que :

- son nom est *nomPointeur* et son adresse est *&nomPointeur*,
- son contenu est indéterminé (valeur d'adresse NULL).

ptReel → (NULL)



## 15.3. INITIALISATION D'UNE VARIABLE POINTEUR

Après avoir déclaré une variable de type pointeur, la première chose à faire, ABSOLUMENT, est d'**initialiser le pointeur**, c'est-à-dire de lui donner un contenu (qui doit être une adresse RAM). Utiliser un pointeur non initialisé (pointant sur NULL) provoque une grave erreur mémoire.

L'exemple suivant montre comment initialiser un pointeur *pti* en lui affectant l'adresse *&i* d'une autre variable *i*. Un pointage est généré du pointeur *pti* vers la case RAM de *i*. La case pointée est alors *i* ou *\*pti* (il est possible d'y accéder par les 2 variables *i* ou *\*pti*) :

```
/*****************************************************************************  
 * INITIALISATION de VARIABLES de TYPE POINTEUR  
 */  
  
int main()  
{    float      *pti;          // pointeur sur un float  
    float      i;             // simple float  
  
    pti= &i;                // pointeur initialisé par pointage sur l'espace RAM d'une variable de type simple  
  
    *pti= 0.0;   // modification de la valeur pointée par 2 accès différents  
    i= 0.0;  
}
```

The diagram shows two memory boxes. The top box contains a pointer variable *pti* with an arrow pointing to a memory cell containing a hexagonal pattern. Below it is another pointer variable *&i* with an arrow pointing to a memory cell containing the letter *i*. The bottom box contains a variable *i* with an arrow pointing to a memory cell containing the value *0.0*. Arrows from both *pti* and *&i* point to the same *0.0* cell, indicating they both point to the same memory location.

**Attention ! Un pointeur doit toujours être initialisé !** L'exemple suivant montre des erreurs d'utilisation de pointeurs, à ne pas reproduire.

```
/*****************************************************************************  
 * ERREURS avec INITIALISATION de VARIABLES de TYPE POINTEUR  
 */  
  
int main()  
{    float      *pti,*ptj;        // pointeurs sur float  
    float      i;             // simple float  
  
    *pti= 0.0;   // ERREUR ! POINTEUR NON INITIALISÉ  
  
    pti= &i;                // initialisation pointeur  
    *pti= 0.0;   // initialisation valeur pointée  
  
    *ptj= *pti;  // ERREUR ! POINTEUR NON INITIALISÉ  
  
    ptj= pti;    // initialisation pointeur  
}
```

The diagram shows three memory boxes. The top box contains two pointers *pti* and *ptj*, each with an arrow pointing to a memory cell containing a hexagonal pattern. The middle box contains a pointer *&i* with an arrow pointing to a memory cell containing the letter *i*. The bottom box contains a variable *i* with an arrow pointing to a memory cell containing the value *0.0*. Arrows from both *pti* and *ptj* point to the same *0.0* cell, indicating they both point to the same memory location. An arrow from *ptj* also points to the *0.0* cell, showing it is also pointing to the same memory location as *pti*.



## FICHE 16 : FONCTION AVEC E/S



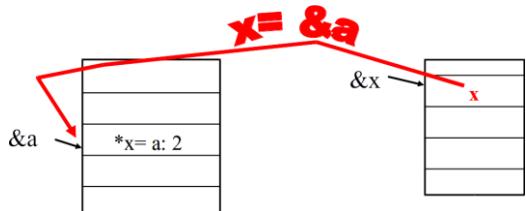


## CREER des FONCTIONS avec PARAMETRE en E/S

Ressources :

[Diapos ORGANISER](#)

Objectif :



- ✓ Comment passer un paramètre par adresse à une fonction ?
- ✓ Comment écrire un paramètre pointeur ?
- ✓ Comment écrire un paramètre tableau ?

### 16.1. FONCTION AVEC PARAMETRE EN E/S (POINTEUR)

Une fonction possède ses propres données locales **privées** et n'a pas accès aux données des autres fonctions. De plus, une fonction ne peut renvoyer qu'un seul résultat.

*Alors, comment transmettre les deux valeurs de signe et abs à la fonction appelante ?*

Les fonctions doivent s'échanger des données : **un paramètre en Entrée/Sortie permettra, ici, de transmettre une deuxième valeur à la fonction appelante ; la valeur de abs doit être récupérée par la fonction appelante avec ce nouveau mécanisme.**

```
*****
FONCTION AVEC deux RESULTATS en SORTIE
*****
```

**ALGO** *int Vabs(int a\_f)*

**VAR** signe, abs: entier signé long // *signe(a\_f) & |a\_f|*  
**DEBUT**

SI ( <i>a_f&lt;0</i> )	abs= - <i>a_f</i>
	signe= -1
SINON	abs= <i>a_f</i>
	signe= 1
FSI	
RETOUR(signe,abs)	

**FIN**

```
*****
FONCTION AVEC deux RESULTATS en SORTIE
*****
```

**int Vabs(int a\_f)**

```
{
    int signe, abs; // signe et valeur absolue de a_f
    if (a_f<0)    abs= -a_f;    signe= -1;
    else          abs= a_f;    signe= 1;
    return(signe,abs);
}
```

## a- Syntaxe fonction avec Paramètre Pointeur

Un **PARAMETRE en ENTREE/SORTIE** est déclaré dans les parenthèses de l'en-tête de fonction (au-dessus de sa définition) ; dans l'exemple suivant, le paramètre en E/S est *ptVabs* ; c'est un **pointeur** (donc l'adresse d'une case RAM). Il sert de **variable locale** dans la fonction, mais avec une particularité : il est **initialisé**, lorsque le CPU entre dans la fonction en mode exécution, **avec l'adresse passée dans l'instruction d'appel de cette fonction** ; dans l'exemple ci-dessous le paramètre en E/S, *ptVabs*, est initialisé avec l'adresse de *vabs* (*&vabs*) -lorsque le CPU redirige son exécution dans la fonction *CalculerVabs()*.

### PROTOTYPE et APPEL DE LA FONCTION :

```
*****  
APPEL d'une FONCTION avec PARAMETRE E/S  
*****  
int CalculerVabs(int a_f, int *ptVabs)  
ALGO main  
    VAR vabs, a, signe : entier signé long  
DEBUT  
    AFFICHER("Donner a : ") SAISIR(&a)  
    signe= Vabs(a,&vabs)  
    AFFICHER("vabs : valeur", vabs)  
FIN
```

### DEFINITION DE LA FONCTION :

```
*****  
FONCTION avec PARAMETRE POINTEUR  
*****  
ALGO int CalculerVabs(int a_f, int *ptVabs)  
    VAR signe_f : entier signé long  
DEBUT  
    SI (a_f<0) *ptVabs= -a_f signe_f= -1  
    SINON *ptVabs= a_f signe_f= 1  
    FSI  
    RETOUR(signe_f)  
FIN
```

```
*****  
APPEL d'une FONCTION avec PARAMETRE E/S  
*****  
int CalculerVabs(int a_f, int *ptVabs);  
int main()  
{    int a, signe, vabs;  
    printf("Donner a : "); scanf("%d",&a);  
    signe= Vabs(a,&vabs);  
    printf("\nvabs : %d", vabs);  
}
```

```
*****  
FONCTION avec PARAMETRE POINTEUR  
*****  
int CalculerVabs(int a_f, int *ptVabs)  
{    int signe_f;  
    if (a_f<0) *ptVabs=-a_f; signe_f=-1;  
    else *ptVabs= a_f; signe_f= 1;  
    return(signe_f);  
}
```

Un **paramètre en E/S est donc passé par adresse** : lors de l'exécution de l'instruction d'appel de fonction, une **adresse de donnée (adresse de vabs)** est transmise au paramètre de la fonction (*ptVabs*). C'est comme si la variable utilisée pour passer la valeur (*vabs*) est en lecture/écriture pour la fonction appelée ; la fonction appelée reçoit l'accès direct à la zone RAM de vabs et peut donc modifier son contenu (valeur de la variable vabs), via le pointeur *ptVabs*. L'accès à la zone mémoire de la fonction appelante *main()* pourrait être vu comme une sorte de « **piratage** » mémoire, normalement privée : la zone RAM de *ptVabs* pointe sur la zone RAM de *vabs* et le paramètre *ptVabs* peut alors écrire dans la variable *vabs*.

Il y a juste **transmission d'un droit d'accès à la zone RAM de la fonction appelée pour la fonction appelante** ; les variables *ptVabs* et *vabs* restent locales et privées à leur fonction. Remarque : il est judicieux, au début, de nommer différemment les variables locales de fonctions différentes, afin de ne pas mélanger ses variables. Cela concerne, en particulier, les paramètres formels -dans la définition *ptVabs* - et les paramètres effectifs -dans l'appel *vabs* - des fonctions.

## b- Exécution fonction avec Paramètre Pointeur

Illustration de l'exécution CPU d'un appel de fonction avec paramètre Pointeur pour le code suivant :

```
/*********************  
APPEL d'une FONCTION avec PARAMETRE E/S  
*****  
int CalculerVabs(int a_f, int *ptVabs);  
int main()  
{    int      a, signe, vabs;  
  
    printf("Donner a : ");  scanf("%d",&a);  
    signe= Vabs(a,&vabs);  
    printf("\nvabs : %d", vabs);  
}
```

```
/*********************  
FONCTION avec PARAMETRE POINTEUR  
*****  
int CalculerVabs(int a_f, int *ptVabs)  
{    int signe_f;  
  
    if (a_f<0)    *ptVabs=-a_f; signe_f=-1;  
    else          *ptVabs= a_f; signe_f= 1;  
    return(signe_f);  
}
```

Les étapes d'exécution sont :

1. Exécution de la fonction principale *main()* :

- ◆ Déclaration et saisie de *a* : un espace RAM est réservé pour *a* dans la zone mémoire du *main()* et *a* reçoit -2 du clavier.
- ◆ **Appel fonction avec passage valeur** -2 et **adresse de *vabs* : *CalculerVabs(-2,&vabs)***

2. Exécution de la fonction appelée *CalculerVabs()* :

- ◆ **Initialisation paramètres en E et en E/S :** le CPU initialise *a\_f* avec la valeur de *a* (*a\_f = a = -2*) et **il initialise *ptVabs* avec l'adresse *&vabs* (*ptVabs=&abs*), dans la zone RAM de *CalculerVabs()*** (Figure 40 montre l'initialisation du pointeur effectuée par le CPU lors de l'appel de fonction).

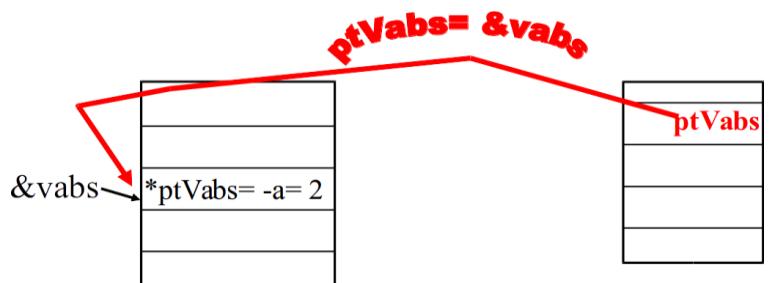


FIGURE 40 : INITIALISATION D'UN PARAMETRE POINTEUR

- ◆ Test du signe de *a\_f* et :

x **affectation de 2 au contenu pointé par *ptVabs* (\**ptVabs*= -(2)), donc écriture directe de 2 dans la case RAM de *vabs* (dans la zone RAM du *main()*).**

x Affectation de -1 à *signe\_f*.

- ◆ Renvoi de la valeur -1 à la fonction *main()*.

3. Fin de l'exécution du *main()*.

- ◆ *signe* récupère la valeur renvoyée par la fonction (*signe* vaut -1).

- ◆ La valeur de *vabs* (2), modifiée pendant l'exécution de la fonction, est affichée.

Remarque : il est possible de déclarer plusieurs paramètres en E/S, même avec des types différents ; les pointeurs doivent être séparés par une virgule dans l'en-tête de définition de fonction. Dans l'instruction d'appel, l'ordre des adresses doit correspondre à l'ordre des paramètres formels.

## 16.2. FONCTION AVEC PARAMETRE TABLEAU

Une fonction possède ses propres données locales **privées** et n'a pas accès aux données des autres fonctions. De plus, une fonction ne peut renvoyer un tableau en Sortie.

*Alors, comment transmettre le contenu d'un tableau, modifié par une fonction, à la fonction appelante ?*

Les fonctions doivent s'échanger des données : **un paramètre en Entrée/Sortie permettra, ici, de transmettre un tableau à la fonction appelante** ; les cases de *tabReel* seront modifiées directement dans la zone RAM de la fonction appelante par la fonction appelée.

```
*****
FONCTION AYANT BESOIN de RENVOYER un TABLEAU
U en SORTIE
*****
```

```
ALGO float RemplirTableau(void)
    VAR tabReel[3]: réels simples précision
DEBUT
    ...
    RETOUR(tabReel)
FIN
```

```
*****
FONCTION AYANT BESOIN de RENVOYER un TABLEAU
AU en SORTIE
*****
```

```
float RemplirTableau(void)
{
    float tabReel[3];
    ...
    RETOUR(tabReel);
}
```

### a- Syntaxe fonction avec Paramètre Tableau

Un **PARAMETRE en ENTREE/SORTIE** de type tableau est déclaré dans les parenthèse de l'en-tête de fonction (au-dessus de sa définition) ; dans l'exemple ci-dessous le paramètre en E/S est *tabReel\_f*.

#### PROTOTYPE et APPEL DE LA FONCTION :

```
*****
APPEL d'une FONCTION avec PARAMETRE E/S
*****
void RemplirTab(float tabReel_f[2])
ALGO main
    VAR tabReel[2] : réel simple précision
        i : entier signé long
DEBUT
    RemplirTab(tabReel)
    POUR i de 0 à 1, pas de 1
        AFFICHER(" valeur",tabReel[i])
    FINPOUR
FIN
```

#### DEFINITION DE LA FONCTION :

```
*****
FONCTION avec PARAMETRE TABLEAU
*****
ALGO void RemplirTab(float tabReel_f[2])
    VAR i : entier signé long
DEBUT
    POUR i de 0 à 1, pas de 1
        tabReel_f[i]= i+1
    FINPOUR
FIN
```

## PROTOTYPE et APPEL DE LA FONCTION :

```
/*
APPEL d'une FONCTION avec PARAMETRE E/S
*/
void RemplirTab(float tabReel_f[2]);
int main()
{
    float tabReel[2]; int i;
    RemplirTab(tabReel);
    for(i=0;i<2;i++)
    {
        printf("\t%.2f",tabReel[i]);
    }
}
```

## DEFINITION DE LA FONCTION :

```
/*
FONCTION avec PARAMETRE TABLEAU
*/
void RemplirTab(float tabReel_f[2])
{
    int i;
    for ( i=0 ; i<=1 ; i=i+1 )
    {
        tabReel_f[i]= i+1;
    }
}
```

Le paramètre *tabReel\_f* est, en fait, un pointeur et nous sommes exactement dans le cas de « **piratage** » mémoire expliqué dans la partie 16.1. En effet, *tabReel\_f* est le **nom du tableau** et représente aussi, en langage C, l'**adresse de la première case du tableau**, autrement écrit *&tabReel\_f[0]*, voir Figure 41.

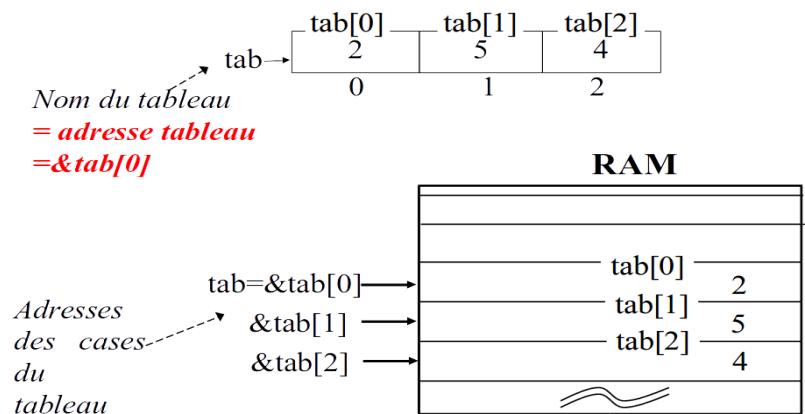


FIGURE 41 : POINTEUR SUR UN TABLEAU

Donc, **le paramètre en E/S, *tabReel\_f*, est ici un pointeur ; il est passé par adresse**. Tout ce qui a été dit dans la partie précédente de ce document s'applique donc ici, en terme de mécanisme d'accès au tableau du main() par le pointeur de la fonction appelée :

1. **La variable paramètre est initialisée, lorsque le CPU exécute l'entrée dans la fonction, avec l'adresse passée dans l'instruction d'appel de cette fonction** ; dans l'exemple ci-dessous le paramètre en E/S, *tabReel\_f*, est alors initialisé avec *tabReel* (adresse du tableau *tabReel*).
2. Lorsque le CPU écrit dans le tableau *tabReel\_f* -dans *RemplirTab()*-, le pointeur *tabReel\_f* écrit directement, en temps réel, dans la zone RAM du tableau *tabReel* -dans *main()*-.

Il y a transmission d'un droit d'accès à la zone RAM de la fonction appelante vers la fonction appelée ; les variables *tabReel\_f* et *tabReel* restent locales à leur fonction et ne sont pas utilisable dans l'autre fonction. Il reste donc judicieux, au début, de **nommer différemment les variables locales de fonctions différentes**, afin de ne pas mélanger ses variables. Cela concerne, en particulier, **les paramètres formels -dans la définition *tabReel\_f*- et les paramètres effectifs -dans l'appel *tabReel*- des fonctions**.

Une chose change par rapport à la première partie : l'écriture de fonctions avec paramètres en E/S de type tableau est plus facile qu'avec des pointeurs simples. En effet, les \* (notation de case pointée) ne sont pas utilisés, il suffit d'utiliser la notation tableau classique. **Le piège à éviter est de bien écrire le seul nom du tableau (sans [ ]) dans l'instruction d'appel de fonction !**

## b- Exécution fonction avec Paramètre Tableau

Illustration de l'exécution CPU d'un appel de fonction avec paramètre Tableau pour le code suivant :

```
/*****************************************************************************  
 * APPEL d'une FONCTION avec PARAMETRE E/S  
******/  
void RemplirTab(float tabReel_f[2]);  
int main()  
{    float tabReel[2]; int i;  
    RemplirTab(tabReel);  
    for(i=0;i<2;i++)  
    {        printf("\t%.2f",tabReel[i]);  
    }  
}
```

```
/*****************************************************************************  
 * FONCTION avec PARAMETRE TABLEAU  
******/  
void RemplirTab(float tabReel_f[2])  
{    int i;  
    for ( i=0 ; i<=1 ; i=i+1 )  
    {        tabReel_f[i]= i+1.;  
    }  
}
```

Les étapes d'exécution sont :

1. Exécution de la fonction principale *main()* :

- ♦ Déclaration de *tabReel* : un espace est réservé pour *tabReel* dans la zone mémoire du *main()*.
- ♦ **Appel fonction avec passage adresse du début du tableau *tabReel* ou *&tabReel[0]* : *RemplirTab(tabReel)*.**

2. Exécution de la fonction appelée *RemplirTab()* :

- ♦ **Initialisation paramètres en E/S : le CPU initialise *tabReel\_f* avec l'adresse *tabReel* (*tabReel\_f = tabReel = &tabReel[0]*), dans la zone RAM de la fonction *RemplirTab()*.**

(Figure 42 montre l'initialisation du pointeur effectuée par le CPU lors de l'appel de fonction).

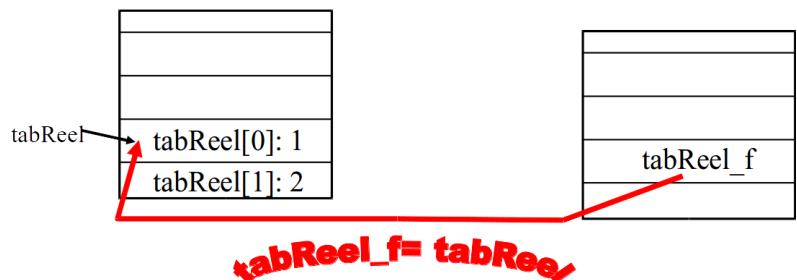


FIGURE 42 : INITIALISATION D'UN PARAMETRE TABLEAU

- ♦ Parcours des cases du tableau *tabReel*, via *tabReel\_f*, et affectation de l'indice *i* augmenté de 1 au contenu pointé par *tabReel\_f* (*tabReel\_f[i] = tabReel[i]*), donc écriture directe de (*i*+1) dans la case RAM de *tabReel[i]* (dans la zone RAM du *main()*).
- ♦ Retour de la fonction appelée.

3. Fin de l'exécution du *main()*.

- ♦ Parcours des cases du tableau *tabReel* du *main()* pour affichage de leur contenu, modifié pendant l'exécution de la fonction appelée.

Remarque : il est possible de déclarer plusieurs paramètres en E/S, même avec des types différents ; les tableaux doivent être séparés par une virgule dans l'en-tête de définition de fonction. Dans l'instruction d'appel, l'ordre des adresses doit correspondre à l'ordre des paramètres formels.