LA GRANDE ÉCOLE
DES TÉLÉCOMMUNICATIONS
ET DES TECHNOLOGIES
DE L'INFORMATION

Filière Smart-ICT

Algorithmique et Programmation C

Mr N.EL FADDOULI
elfaddouli@emi.ac.ma
nfaddouli@gmail.com

Année Universitaire:2024/2025

1

Plan

3

Le Langage C

46

Le langage C: Prérequis

- Notions de base d'Algorithmique:
  - Variable
  - Affectation
  - Lecture des entrées
  - Affichage des résultats
  - Instructions de sélection (*instructions conditionnelle*)
  - Les boucles
- Analyse de problème et développement d'algorithme

47

Le langage C: C Vs Python

- Le langage C est l'un des langages de programmation les plus utilisés
- Il a été classé à la 4ème position dans le top 10 des langages les plus utilisée en 2023.

Python	C
Programmation procédurale, fonctionnelle et orientée objet	Programmation procédurale
Exécuté par un interpréteur ligne par ligne (exécution lente)	Compilé pour avoir un exécutable en langage machine (exécution rapide)
Code plus court que celui en C (prototypage rapide)	
Python est principalement développé en C et plusieurs librairies sont en C	
Déclaration de type n'est pas obligatoire	Déclaration de type obligatoire
Typage dynamique	Typage statique
Gestion automatique de la mémoire	Le programmeur gère lui-même l'allocation et la libération de la mémoire

Le langage C: Structure d'un programme en C (1/2)

- Dans un programme en langage C, on indique les **variables** et les **instructions** qui les manipules pour lire les entrées et calculer les sorties.
- **Exemple**

Algorithme
Entrées : (Aucune)
Sorties : message
Traitement : Affichage de ''Bonjour la 1ere année''
Ecrire ("Bonjour la 1ère année")

```
# include<stdio.h>  /* bibliothèque C pour E/S */
  void  main( )     /*programme principal */
  {
  printf("Bonjour …");     /*affiche le message entre " */
  }
```

{ et } indiquent le début et la fin du programme

/* et */ délimitent les commentaires qui sont ignorés lors de la compilation

; indique la fin d'une instruction

Le langage C: Structure d'un programme en C (2/2)

- Un programme (*fichier d'extension* ***.c***) peut contenir cinq sections dont certaines sont optionnelles organisées comme suit:

> ❶ **Directives de compilation**
>
> ❷ **Déclaration des variables externes (globales)**
>
> ❸ **Déclaration des prototypes de fonctions**
>
> ❹ **Programme principal**
>
> ❺ **Définition des fonctions**

- Les directives de compilation ❶ sont des instructions données au préprocesseur du compilateur. Elles commencent par le symbole **#** et sont traitées **avant** la phase de compilation proprement dite.
- Ces directives permettent de contrôler divers aspects de la compilation, comme l'**inclusion de fichiers**, la **définition de constantes**, et la gestion conditionnelle de blocs de code.
- Le programme principal ❹ contient les instructions exécutées à son lancement .
- Nous verrons les autres sections dans les prochains slides de ce support.

Le langage C: Les directives de compilation (1/2)

- La directive **#include** permet d'inclure le contenu de fichiers dans le programme et qui sont généralement des bibliothèques des fonctions appelées dans le programme.
- La syntaxe utilisée est la suivante: **#include <***nom-fichier***>** ou **#include "***nom-fichier***"**
- Les fichiers entre chevrons **< >** sont recherchés dans les répertoires standard du compilateur.
- Les fichiers entre guillemets " " sont recherchés d'abord dans le répertoire de travail là où il y a le programme, puis dans les répertoires standard.
- Ces bibliothèques sont généralement des fichiers texte dont l'extension est **.h** mais peut être n'importe quelle autre extension (.c par exemple)Exemples:

```
#include<math.h>      /* Fonctions mathématiques */
#include<ctype.h>     /* Fonctions sur des caractères */
#include<string.h>    /* Fonctions de manipulation des chaînes de caractères */
#include<malloc.h>    /* Fonctions de gestion de la mémoire*/
```

Le langage C: Les directives de compilation (2/2)

- La directive **#define** est utilisée pour définir des **constantes** ou des **macros**, qui sont des symboles ou des fragments de code qui seront remplacés par une valeur ou un code spécifique avant la compilation.
- Pour définir une constante on utilise la syntaxe suivante: **#define** *constante valeur*

Exemple: **#define** PI 3.14159

Chaque fois que le symbole PI est rencontré dans le programme, lors de la compilation, il est remplacé par 3.14159

- Pour définir une **macro** on utilise la syntaxe suivante:

#define ***macro***(*paramètres*) *code_de_remplacement*

Exemple:
```
#define Somme(a, b)   (a + b)
#define Max(a , b)    (a > b ? a : b )
```

Chaque fois que la macro Somme ou Max est rencontrée dans le programme, lors de la compilation, elle est remplacée par le code associé. Par exemple Max(x, 4) sera remplacé par (x > 4 ? x : 4)

- Il y a d'autre directives de compilation comme les directives **conditionnelles** que nous verrons dans des exemples par la suite.

Le langage C: Le programme principal

- Le programme principal le langage C est une fonction **main()** ayant la structure minimale suivante:

```
main() { ……. /* variables */
         ……. /* instructions */
       }
```

- Chaque instruction ou déclaration de variables se termine par point-virgule.
- On peut également spécifier la valeur de retour (entier par exemple) de la fonction et des paramètres:

```
int main(char args[ ]) {
    ……. /* variables */
    ……. /* instructions */
return entier;   /* on indique la valeur de retour, par
                    exemple return 1; */
}
```

On n'indique pas le type de retour ou on utilise **void** si la fonction n'a pas de valeur de retour.

Le langage C: Les variables (1/3)

- Les variables sont des **symboles** représentant des données en **entrée** ou **calculées**.
- Chaque variable est caractérisée par:

  - Un **nom** qui l'identifie dans le programme et qui contient des lettres, des chiffres et le caractère "_" et commence par ce dernier ou une lettre. On n'utilise pas des mots clés (if, while, …)

  - Une **valeur** à un instant donnée. Elle peut être remplacée par une autre valeur.
  - Un **type** qui représente la nature de la valeur.

  - Une **zone mémoire** dans la RAM qui contient la valeur.

- Pour déclarer une variable dans le programme, on utilise la syntaxe suivante:

  type ***nom_variable1, nom_variable2,*** …. **;**

  Il y a plusieurs types possibles: **int** (*entier*), **long** (*entier long*), **float** (*réel*), **double** (*réel à double précision*), **char** (*un caractère*), ….

- **Exemple: main() {** float a, prix;
  int A, B=9;
  char rep; …. **}**

- Pour les noms de variables, le langage C fait la distinction entre minuscule et majuscule
- La valeur initiale d'une variable est **indétermnée** (dépend du contenu de la zone mémoire réservée pour la variable)

Le langage C: Les variables (2/3)

- La taille de la zone mémoire d'une variable dépend de son type.

Type	Signification	Taille en octet (généralement)	Valeurs limites
int , short	Entier	2	32768 à 32767
long	Entier	4	-2147483648 à 2147483647
float	Réel	4	-10^{-37} à 10^{38}
double	Réel à double précision	8	-10^{-307} à 10^{308}
long double	réel long	10	-10^{-4932} à 10^{4932}
char	caractère	1	
unsigned	entier positif	2	0 à 65535

- La taille dépend aussi du système d'exploitation et du compilateur utilisé
- La fonction **sizeof(***type, variable ou valeur***)** retourne la taille en **octets** du paramètre donné.

```
int n, i ;   n = sizeof (i);      /* n reçoit la taille en octet de i (int) */
n = sizeof (int);                 /* n reçoit la taille en octet d'un entier*/
n = sizeof (3.14);                /* n reçoit la taille en octet pour stocker 3.14 */
```

Le langage C: Les variables (1/3)

- Avant la norme **C99** (*C89/C90*) toutes les déclarations devaient être placées en premier dans un bloc avant les instructions.

Exemple: main() {
```
        /* Toutes les déclarations de variables en haut */
        int  x = 5, y;
        float r;
        /*  Les instructions ensuite */
        ….
}
```

Bloc de code:
{ /*Déclaration*/
….
/*Instructions*/
….
}

- À partir de la version **C99** du langage C (*norme ISO/IEC 9899:1999*) on peut mélanger les **déclarations** et les **instructions** dans le même bloc de code.

Exemple: main() {
```
        /* déclarations */
        int  x = 5, y;
        /* instructions */
        ….
        /* déclarations */
        float r;
….              }
```


Le langage C: L'affectation (1/2)

- L'affectation est une instruction qui consiste à **évaluer** (*calculer*) la valeur d'une **expression** et la stocker dans une **variable**.
- La syntaxe de l'affectation est la suivante: ***variable* = *expression* ;**

Exemple: int i ;
```
        float  j = 3.5 ;  /* Déclaration et initialisation */
        char  c = 'A' ; /* Les valeurs de types caractère sont mises entre simple quottes */
        i = 36;
        i = i *2 + 6 ;
```

- Une **conversion implicite**, si elle est possible, est effectuée lorsque le type de la valeur de l'expression est différent de celui de la variable.

Exemple:
```
i = j ;    /*  i reçoit la partie entière 3 du réel j  */
i = c ;    /*  i reçoit 65 le code ascii de la lettre  'A'  */
c = 66;    /*  c contiendra  le caractère dont le code ascii est 66 ⇔ c =  'B';  */
```

- **N.B: Une erreur est produite lorsque la conversion implicite est impossible**

Le langage C: L'affectation (2/2)

- On peut faire une **conversion forcée** (**cast**) d'une expression vers un type spécifique en utilisant la syntaxe suivante:

(*Nouveau_Type*) *Expression*

Exemple:

```
int i = 9
float y , j = 2.5;
y =  int ( j ) * i ;  /* y reçoit  18.0 */
y = j * i;            /* y reçoit  22.5 */
```

- Il est possible de **perdre des données**, en particulier lors de la conversion de types plus grands en types plus petits (par exemple, un double en int ou un long en char).
- Lors de la conversion d'un type plus grand en un type plus petit, si la valeur ne peut pas être représentée par le nouveau type, il peut y avoir des comportements imprévisibles ou des dépassements de capacité (**overflow**).

Le langage C: Les opérateurs arithmétiques

- Les opérateurs de calcul utilisés dans les expressions arithmétiques sont les suivants:

Opérateur	Calcul effectué	Exemple int A=19, B=7,C; float D=12, H;
+	La somme de deux nombres	C = A + B;
-	La différence entre deux nombres	C = A-B;
*	Le produit de deux nombres	C = A*B;
/	- Résultat de la division réelle d'un nombre par un autre **si l'un des deux opérandes est un réel.** - Le quotient de la division entière (euclidienne) d'un **entier** par un autre.	H = D/5 ; ***/* H reçoit 2.4 */*** C = A/B; ***/* C reçoit 2 */*** H= **(float)**A/B; ***/* H reçoit 2.714285 */***
%	Le reste de la division entière d'un **entier** par un autre	C = A%B; ***/* C reçoit 5*/***

Le langage C: L'affectation combinée

- On peut combiner une affectation et un opérateur arithmétique lorsqu'on veut faire un calcul sur une variable et mémoriser le résultat dans cette même variable:

  variable *opérateur*= ***expression ;***

  Où opérateur peut être: **+, -, *, / ou %**

- **Exemple**

```
main () {  int i = 9, j=2;
   i += j  ;        /* ⇔  i = i + j; */
   i /= j  ;        /* ⇔  i =  i / j; */
   i -= j  ;        /* ⇔  i =  i- j; */
   …..
}
```


Le langage C: Les opérateur d'incrémentation/décrémentation (1/2)

- Pour incrémenter ou décrémenter un nombre de **1**, on utilise respectivement les deux opérateurs unaires **++** et -- comme suit:

  ++*variable* **;** *ou* ***variable*++ ;**

  --*variable* **;** *ou* ***variable*-- ;**

- Exemple:

```
main () {  int   i = 9 ,  j ;
   i++ ;   /* ou ++i   ⇔ i = i + 1 */
   --i ;   /*  ou i--   ⇔ i = i - 1 */
   …….
}
```

Le langage C: Les opérateur d'incrémentation/décrémentation (1/2)

- L'instruction d'incrémentation/décrémentation peut être combinée à une autre instruction comme l'affectation. On aura donc deux instructions à exécuter dans l'ordre en commençant par l'incrémentation/décrémentation ou l'autre instruction.
- On aura une **post-incrémentation** ou une **pré-incrémentation**. La différence réside dans le moment où la valeur de la variable est modifiée par rapport à l'utilisation de cette valeur dans l'expression.
- **Exemple**: int i = 5 , j ;

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

On doit faire une affectation de la valeur de i à j et incrémenter i. Il s'agit là d'une **post-incrémentation**. On aura donc ces deux instructions dans cet ordre: i++; j = i; ⇨. i=6 et j=6

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

On a une **pré-incrémentation**. On aura donc ces deux instructions dans cet ordre: j = i; i++; ⇨. i=6 et j=5

Le langage C: Les opérateur de traitement de bits

- les **opérateurs de traitement de bits** (ou **opérateurs bitwise**) permettent de manipuler directement les bits qui composent les variables entières (int, long, char, etc.).
- Ils sont souvent utilisés pour des opérations bas niveau, comme la manipulation de drapeaux (flags) ou l'optimisation de certains algorithmes.

Opérateur	Nom	Description
&	**ET** bit à bit (AND)	Renvoie 1 si les deux bits sont 1, sinon renvoie 0.
|	**OU** bit à bit (OR)	Renvoie 1 si l'un des deux bits est 1, sinon renvoie 0.
^	OU exclusif (**XOR**)	Renvoie 1 si un seul des bits est 1, sinon renvoie 0.
~	**NON** bit à bit (NOT)	Inverse tous les bits (0 devient 1 et 1 devient 0).
<<	Décalage à gauche	Décale les bits vers la gauche et ajoute des 0 à droite.
>>	Décalage à droite	Décale les bits vers la droite et ajoute des 0 à gauche (ou le bit de signe pour les nombres signés).

Le langage C: Les opérateur de traitement de bits

☞ **Exemple**:

```
int i = 597;    /* (597)10 =(0000 0000 0010 0101)2  */
int j = 12;     /* (12)10  =(0000 0000 0000 1100)2  */
int k;
k = i & j ;     /* k  =     (0000 0000 0000 0100)2 = (4)10  */
k = j >> 2 ;    /* k =      (0000 0000  0000 0011)2 = (3)10  */
k = ~j ;        /* k =      (1111 1111 1111 0011)2    complément à 1 de j   */
```

☞ **Exemple:** Utilisation de drapeaux (flags) pour gérer des permissions

```
#define FLAG_READ    1    /* 0000 0001   permission de lecture (1er bit) */
#define FLAG_WRITE   2    /* 0000 0010   permission d'écriture (2ème bit)*/
#define FLAG_EXEC    4    /* 0000 0100   permission d'exécution (3ème bit)*/
main() {
int  per = 0;  /* Aucune permission */
per |= FLAG_READ;     /* Activer la permission de lecture */
per |= FLAG_WRITE;    /* Activer la permission d'écriture */
per &= ~FLAG_WRITE; /* Désactiver la permission d'écriture */
….. }
```

per = per | FLAG_READ;

Le langage C: L'affichage des sorties (1/2)

☞ Il y a plusieurs fonctions qui permettent d'afficher des sorties sur écran.

☞ On utilise généralement la fonction **printf()** définie dans la bibliothèque standard **stdio.h** (*Standard Input Output*) et qui permet d'afficher du **texte**, des **variables**, et des **résultats** de calculs dans la console.

printf("*chaîne de formatage*", *variables*, *expressions*);

☞ La chaîne de formatage contient du **texte** et des **spécificateurs** de format

☞ **Exemple**:

```
void main() { int   A=34, N=5;  float prix=34.6;  char  O='G' ;
   printf("Bonjour…");
   printf("les deux entiers %d  et %d \n", A, B);
   printf("Le total est %f  \n", N*prix);
   printf("Le prix unitaire %.2f \n", prix) ;
   printf("\n \t Valeur de A= %d", A++);
   printf("La caractère est %c  \n", O);  }
```

Deux chiffres après la virgule. Par défaut, printf() affiche **six** chiffres après la virgule.

On a une **prét-incrémentation**. On aura donc ces deux instructions dans cet ordre:
printf("Valeur de A= %d\n", A); A++;
⇨ Ecran: Valeur de A=34 A=35

Le langage C: L'affichage des sorties (2/2)

- Les spécificateurs de format sont:

Spécificateur	Description	Exemple
%d	Entier (int)	int x = 10; printf("%d", x);
%ld	Entier long	long l = 123456; printf("%ld", l);
%f	Réel (float)	float y = 3.14; printf("%f", y);
%lf	Réel (double))	double y = 3.14; printf("%lf", y);
%c	Caractère	char c = 'A'; printf("%c", c);
%s	Chaîne de caractères (string)	char str[10] = "Bonjour"; printf("%s", str);
%x	Format hexadécimal	int a = 255; printf("%x", a);
%%	Signe de pourcentage	printf("100%% correct");
%e	Format exponentiel	float r= 12345.6789; printf("%e", r);
%p	Adresse mémoire d'un pointeur	int* p1; printf("%p", p1);

- On peut également préciser des **caractères de contrôle** de l'affichage en utilisant **\n** *(retour à la ligne)*, **\t** *(faire une tabulation)*, **\a** *(envoyer un bip sonore)*, **\"** *(afficher le guillemet)*, **\b** *(reculer le curseur d'un pas)*

Le langage C: Le lecture des entrées(1/2)

- Il y a plusieurs fonctions qui permettent lire les entrées à partir du périphérique d'entrée standard (clavier) et les stocker dans des variables.
- On utilise généralement la fonction **scanf()** définie dans la bibliothèque standard **stdio.h** et qui permet lire des valeurs de différents types dans des variables.

scanf(" spécificateurs de format ", adresses de *variables*);

- Les spécificateurs de format permettent de spécifier les types des valeurs attendues.
- Pour avoir l'adresse d'une variable, on utilise l'opérateur **&** : **&variable**
- **Exemple**:

```
void main() {  int  N; float prix; char  O ;
   printf("Donnez le nombre d'articles: ");
   scanf("%d", &N) ;
   printf("Donnez le prix unitaire: ");
   scanf("%f", &prix) ;
   printf("Montant à payer: %.2f \n", N*prix) ;  }
```

```
printf("Donnez le nombre d'articles et le prix: ");
scanf("%d%f", &N, &prix) ;
```


Le langage C: Le lecture des entrées(2/2)

- Lorsqu'on utilise **scanf()** pour lire un caractère (**%c**) après une autre entrée numérique (entier ou réel), un caractère de nouvelle ligne (\n) laissé dans le **tampon d'entrée** peut poser un problème.

Exemple:

```
void main() { int  a; char r;
    printf("Donnez un entier:"); scanf("%d",&a);
    printf("Donnez un caractère:"); scanf("%c", &r);
    printf("\n %d    %c\n", a, r);
}
```

```
Donnez un entier:123
Donnez un caractere:
123
```

L'utilisateur n'aura pas la main que pour saisir la valeur de r ⇨ Le 2ème scanf stockera le caractère \n (code ascii 10) automatiquement dans r.

- Pour contourner cela, on peut:
  - ajouter un espace avant %c dans scanf() pour ignorer les espaces et sauts de ligne: scanf(" %c", &o)
  - faire appel à **fflush(stdin)** avant scanf("%c", &o) afin de vider le tampon de lecture utilisé par scanf.

