



GRANDE ÉCOLE D'INGÉNIEUR GÉNÉRALISTE

Programmation Embarquée

2021-2022

www.hei.fr

A propos de moi

- Ingénieur télécommunications et réseaux
 - Master 2 informatique.
Spécialité: informatique mobile et répartie
 - Doctorant à HEI entre 2015 et 2018.
Spécialité: réseaux de capteurs
 - Enseignant chercheur en informatique
Département Computer Science & Maths
Bureau T336
-
- HEI 3 – ISEN 3 ème année: Objets connectés
 - HEI 4 ITI: Programmation embarquées – Réseaux Ubiquitaires
 - HEI 4 MOIL: IoT et traçabilité
 - HEI 5: Méthodologie de Recherche

A propos du cours: Modalités et Evaluation

21h cours-travaux dirigés:

18 avec moi même

3h avec un intervenant extérieur



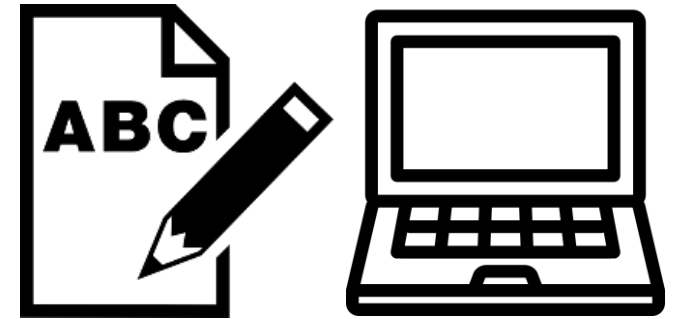
Classe
Entière



Appel
Obligatoire



Ordinateur
Obligatoire



Exercices sur Papier
et Ordinateur

A propos du cours: Contenu, Modalités et Evaluation

18h Travaux Pratiques:

12h de TP Programmation C – 6 TPs en total : 5 TPs d'entraînement – 1 TP noté

6h intervenant extérieur



½ Classe



Appel
Obligatoire



Ordinateur
Obligatoire



Matériel fourni
Si nécessaire

A propos du cours: Contenu, Modalités et Evaluation

9h de Projet



Evaluation
de groupe



Appel
Obligatoire



Ordinateur
Obligatoire

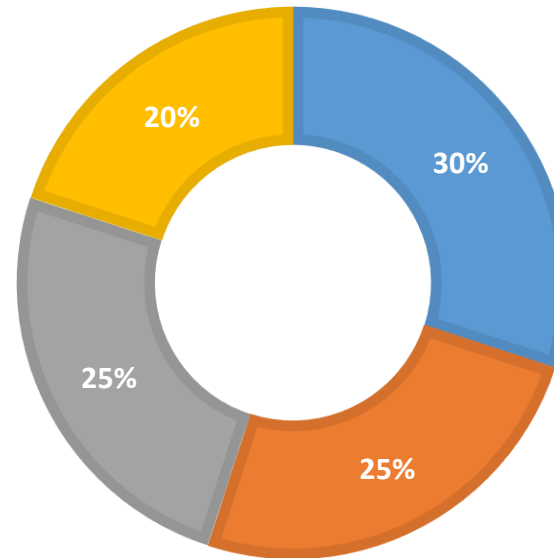


Matériel fourni
Si nécessaire

A propos du cours: Contenu, Modalités et Evaluation

Evaluation

■ TP noté - Pop Quiz ■ Exament Final Individuel
■ Projet En Groupe ■ Réseaux ubiquitaires



A propos du cours: références

- Langage C – David Dubois – Décembre 2017
- Programmation en Langage C - Alexandre Meslé - 12 mars 2019
- Head First C – David Griffiths & Dawn Griffiths

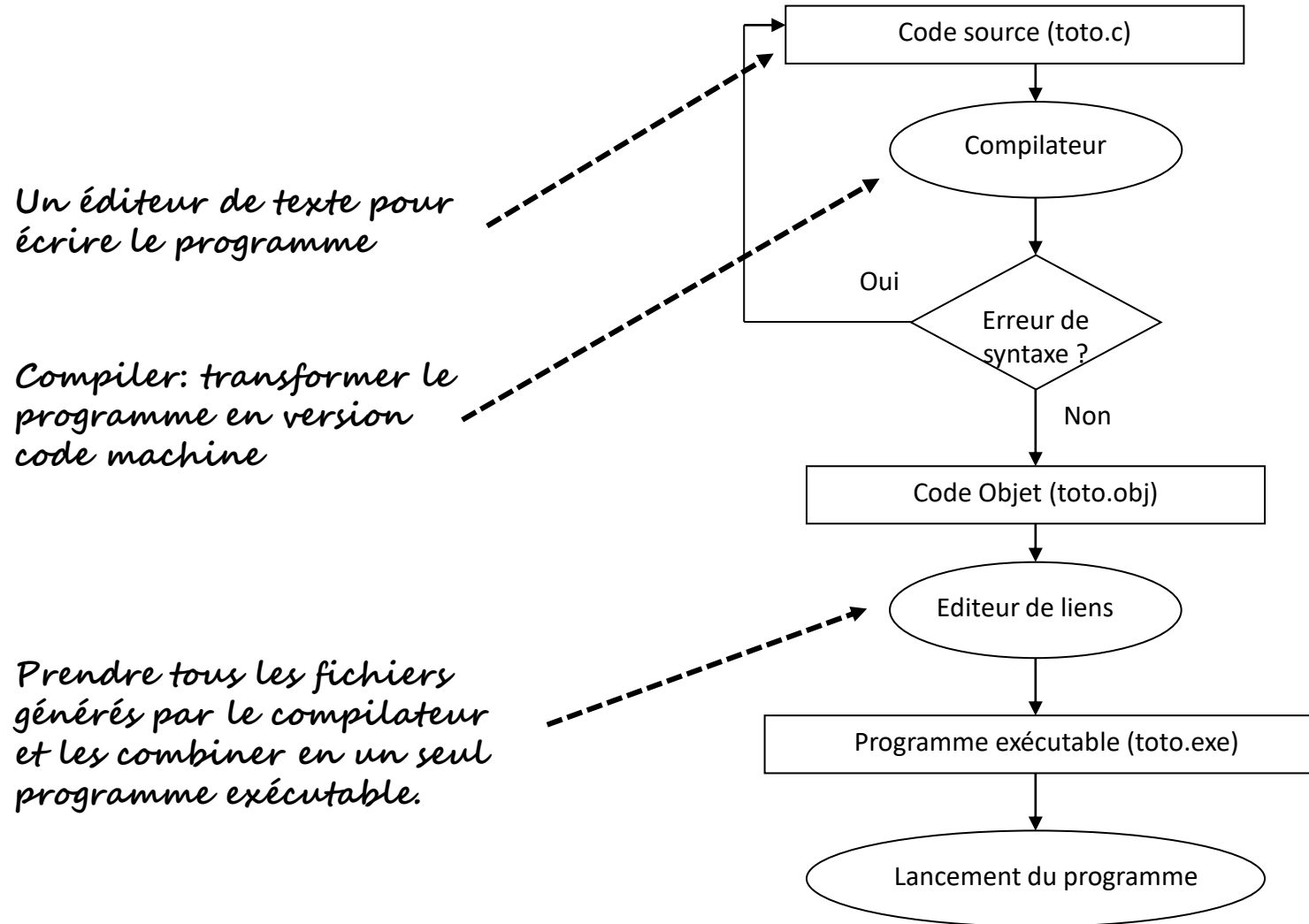
Sommaire

- I. Introduction
- II. Entrées et Sorties
- III. Opérateurs
- IV. Traitements conditionnels et boucles
- V. Tableaux et Structures
- VI. Fonctions
- VII. Pointeurs



I. Introduction

1. Programmer en C



I. Introduction

2. Structure d'un programme C

Importation des bibliothèques: `#include <fichier>`

Les informations dont le compilateur a besoin pour les fonctions prédéfinies (`printf`, ...) se trouvent dans des fichiers d'en-tête (header).

Selon ce que l'on souhaite faire dans notre programme, on peut avoir besoin de différentes fonctions.

Celles-ci sont disponibles dans des bibliothèques:

`math.h` : contient les déclarations des fonctions mathématiques,

`stdio.h` : contient les déclarations des fonctions standards d'entrée sortie,

`stdlib.h`: contient les déclarations des fonctions 'Memory Allocation / Freeing'.

Etc...

Bloc d'un programme:

Une accolade ouvrante "{" et une accolade fermante "}" constituent un bloc.

Un programme en langage C peut contenir un nombre quelconque de blocs.

Ceux-ci renferment toutes sortes d'instructions. Ces blocs peuvent être aussi inclus les uns dans les autres.

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

```
int main ( )
```

```
{
```

```
/*
```

```
Ceci est un commentaire :
```

```
L'instruction ci-dessous
```

```
affiche "Hello world !"
```

```
Ces phrases sont ignorées par
```

```
le compilateur
```

```
*/
```

```
printf ( "Hello world ! " ) ;
```

```
return 0 ;
```

```
}
```

Remarquez que chaque ligne se termine par un point-virgule.

I. Introduction

2. Structure d'un programme C

```
#include <stdio.h>

int main ( )
{
    /*
    Ceci est un commentaire :
    L'instruction ci-dessous
    affiche "Hello world !"
    Ces phrases sont ignorées
    par le compilateur
    */
    printf ( "Hello world ! " );

    return 0 ;
}
```

Tous les codes C fonctionnent à l'intérieur des fonctions. La fonction la plus importante que vous allez trouver dans n'importe quel programme C est la fonction `main()`.

C'est le point de départ de tout le code de votre programme!

`int main()` signifie que notre fonction doit retourner un entier à la fin de l'exécution, on fait ça en retournant 0 (zero) à la fin du programme.

0 signifie que le programme est bien exécuté.

I. Introduction

2. Structure d'un programme C

Commentaires:

Un commentaire est une séquence de caractères ignorées par le compilateur, on s'en sert pour expliquer des portions de code.

Variables:

Une variable est un emplacement de la mémoire dans lequel est stockée une valeur. Chaque variable porte un nom et c'est ce nom qui sert à identifier l'emplacement de la mémoire représentée par cette variable.

Pour utiliser une variable, la première étape est la **déclaration**.

Si on souhaite **affecter** une valeur à la variable, on utilise l'opérateur '='
Saisie et **affichage** à voir ultérieurement.

Constantes:

Une constante est une valeur portant un nom, contrairement aux variables, elles ne sont pas modifiables.

Les constantes en C sont non typées, on les définit dans l'entête de la source, juste en dessous des **#include**.

La syntaxe est **#define <NOM CONSTANCE> <valeurConstante>**

```
#include <stdio.h>
#define N 25
```

```
int main ( )
{
int v1,v2;
/*
```

```
Ceci est un commentaire :
L'instruction ci-dessous
affiche "Hello world !"
Ces phrases sont ignorées par
le compilateur
*/
```

```
printf ( "Hello world ! " ) ;
```

```
return 0 ;
}
```

On peut avoir des constantes de type:

- Décimal (premier chiffre non nul). E.g. 12
- Octal (doit commencer par un zéro). E.g. 014
- Hexadécimal (doit commencer par 0x ou 0X). E.g. 0x90, 0XC

I. Introduction

2. Structure d'un programme C

Types de données:

- Type de données entières:

short (2 octets), **int** (2 ou 4 octets), **long** (4 ou 8 octets).

Si on veut stocker la donnée correspondante avec ou sans signe, on peut compléter ces types de données avec la clause **signed** (signé) ou **unsigned** (non signé). Si ces clauses sont omises, les types de données sont signés par défaut.

- Type char (caractère): utilisé pour représenter un caractère. **char** (1 octet)
- Types de données réelles: Les nombres décimaux en langage C sont dits à virgule flottante : Ils représentent des valeurs non entières et sont des approximations des nombres réels. Le langage C dispose de 3 types:

float (4 octets) , **double** (8 octets) et **long double** (10 octets)

```
#include <stdio.h>
#define N 25

int main ( )
{
    char c;    /* déclaration d'une
                variable char */
    int i= 4;   /* la variable I prend la
                valeur 4 */
    double d= 15.4; /* la variable d
                    prend la valeur 15.4 */
    return 0 ;
}
```

Remarquez que chaque ligne se termine par un point-virgule.

I. Introduction

2. Structure d'un programme C

Pour information:

Type	Occupation mémoire (octet)	Plage de valeur
char	1	-128 ... 127
unsigned char	1	0 ... 255
int	2 (ou 4*)	-32 768 ... 32 767
unsigned int	2 (ou 4*)	0 ... 65 535
short	2	-32 768 ... 32 767
unsigned short	2	0 ... 65 535
long	4	-2 147 483 647 ... 2 147 483 648
unsigned long	4	0 ... 4 294 967 295
Type à virgule flottante		
float	4	$3,4 \cdot 10^{-38} \dots 3,4 \cdot 10^{38}$
double	8	$1,7 \cdot 10^{-308} \dots 1,7 \cdot 10^{308}$
long double	10	$3,4 \cdot 10^{-4932} \dots 3,4 \cdot 10^{4932}$

* dépend du processeur de l'ordinateur

II. Entrées et Sorties

Pour transmettre des données saisies au clavier à un programme (entrées) ou pour afficher à l'écran les données par un programme (sorties), il faut faire appel à un ensemble de fonctions appartenant à la bibliothèque d'entrée-sortie. Il faut donc faire apparaître en début de programme l'instruction suivante :

#include <stdio.h>

Sorties: printf(), putchar ()

1. printf ():

printf("string_format",[argument_1, argument_2,..., argument_n]) ;

Affichage formaté	
%d	Entier int
%u	Nombre entier non signé
%o	Nombre entier octal
%x ou %X	Nombre entier hexadécimal
%c	Caractère ASCII
%f	Nombre à virgule flottante
%e ou %E	=%f + format exponentiel
%s	Chaîne de caractères

```
#include <stdio.h>
int main ( )
{
    int a , b , c ;
    a = 1 ;
    b = 2 ;
    c = 3 ;
    printf ("La valeur de a est %d ,
    celle de b est %d , et celle de c
    est %d . " , a , b , c ) ;
    return 0 ;
}
```

Affiche:

La valeur de a est 1, celle de b est 2, et celle de c est 3.

La fonction **printf** traite aussi les caractères non affichables (séquences d'échappement) qui consistent à effectuer une action (saut de ligne, tabulation...)

\n : saut de ligne

\t : tabulation horizontale

\b : retour du curseur

Etc...

II. Entrées et Sorties

Sorties: printf(), putchar()

2. putchar ():

putchar affiche un caractère sur le périphérique de sortie standard (normalement l'écran). La donnée à affiché est écrite sous forme de paramètre entre les parenthèses de **putchar**.

```
#include <stdio.h>
int main()
{
    putchar('c');
    putchar(97); //code ASCII du caractère a
    return 0;
}
```

Affiche:

ca

'N'oubliez pas les cotes'

*On peut remplacer la constante
<< caractère >> elle-même par le code ASCII du caractère entre les parenthèses*

II. Entrées et Sorties

Entrées: scanf (), getchar(), getch(), getche()

1. scanf():

```
scanf("string_format", [argument_1, argument_2, ..., argument_n]) ;
```

L'opérateur d'adressage & appliqué au nom de la variable informe la fonction **scanf** de l'emplacement de stockage de la valeur saisie. **scanf** attend comme argument non pas le nom (donc la valeur) d'une variable, mais au contraire son adresse.

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

L'instruction ci-dessus permet de transmettre à la fonction **scanf** une valeur saisie au clavier et qui sera rangée dans la variable **x**. La spécification de format **%d** dit que cette valeur doit être un entier. Naturellement, la variable **x** doit avoir été préalablement définie et son type doit être compatible avec le format.

```
#include <stdio.h>
int main ( )
{
    int i;

    printf("Saisir une
valeur\n");
    scanf("%d", &i);
    printf( "la valeur est:
%d", i);
    return 0 ;
}
```

Affiche:

Saisir une valeur: 3

La valeur est: 3

Rappel:

Affichage formaté	
%d	Entier int
%u	Nombre entier non signé
%o	Nombre entier octal
%x ou %X	Nombre entier hexadécimal
%c	Caractère ASCII
%f	Nombre à virgule flottante
%e ou %E	=%f + format exponentiel
%s	Chaîne de caractères

II. Entrées et Sorties

Entrées: scanf (), getchar(), getch(), getche()

2. getchar ():

Pour la lecture des caractères isolés (type **char**) depuis le périphérique d'entrée standard (clavier), on utilise la macro **getchar**. Celle-ci lit un caractère isolé depuis le clavier et le met à la disposition du programme. Après l'instruction **getchar()**; le programme attend une saisie de l'utilisateur. L'instruction lit un caractère, mais ne s'occupe pas de sa mémorisation. La saisie effectuée par cette instruction sera donc perdue. Pour mémoriser le caractère lu, il faut l'affecter à une variable de type **char** ou **int**.

3. La fonction **getche** lit un caractère isolé et en plus affiche ce caractère lu (écho). La fonction **getch** lit un caractère sans l'afficher. Les déclarations des deux fonctions se trouvent dans le header **conio.h**.

```
#include <stdio.h>
int main()
{
    printf("%c", getchar());
    return 0;
}
```

L'utilisateur rentre: c

Affichage:
c

II. Entrées et Sorties

Entrées: scanf (), getchar(), getch(), getche()

2. getchar ():

Essayez l'exemple suivant, que constatez vous?

```
#include <stdio.h>
#include <conio.h>
int main()
{
    char c,d;
    c=getchar();
    d=getchar();
    printf("%c %c", c,d);
    return 0;
}
```

II. Entrées et Sorties

Entrées: scanf (), getchar(), getch(), getche()

2. getchar ():

Essayez l'exemple suivant, que constatez vous?

```
#include <stdio.h>
#include <conio.h>
int main()
{
    char c,d;
    c=getchar();
    d=getchar();
    printf("%c %c", c,d);
    return 0;
}
```

Le deuxième caractère qui correspond au deuxième getchar () n'est pas affiché à l'écran!

Solution:

getchar() renvoie le premier caractère dans le tampon d'entrée, puis le supprime du tampon d'entrée.

Mais les autres caractères sont toujours dans la mémoire tampon d'entrée (\n ou espace dans votre exemple).

Vous devez effacer le tampon d'entrée avant d'appeler getchar() à nouveau.

Soit en utilisant fflush(stdin) pour vider le tampon soit en appelant getchar() une fois en plus qui va récupérer alors le caractère \n

II. Entrées et Sorties: TD 1

1. Ecrire un programme qui affiche un message sur l'écran.
2. Ecrire un programme qui affiche un entier sur l'écran sans l'initialiser en utilisant l'affichage format.
3. Ecrire un programme qui définit une variable i, l'initialise avec la valeur 5 puis l'affiche sur l'écran.
4. Ecrire un programme qui définit une variable char et l'initialise, puis l'affiche sur l'écran.
5. Ecrire un programme qui lit un caractère via getchar et l'affiche via putchar
6. Décrire l'affichage du programme suivant:

```
#include<stdio.h>
int main ( ) {
char c;
printf("Entrez un caractere.\n");
printf("Validez la saisie par <Entree>\n");
c=getchar();
printf("Le caractere lu est %c. \n",c);
printf("Entrez encore un caractere:\n");
c=getchar();
printf("Le caractere est: %c \n",c);
return 0;
}
```

III. Opérateurs

Opérateur	Signification	Exemple
Opérateurs de comparaison		
==	Opérande 1 égal à opérande 2 ?	a==b
!=	Opérande 1 différent de opérande 2 ?	a !=b
<=	Opérande 1 inférieur ou égal à opérande 2 ?	a<=b
>=	Opérande 1 supérieur ou égal à opérande 2 ?	a>=b
<	Opérande 1 inférieur à opérande 2 ?	a	Opérande 1 supérieur à opérande 2 ?	a>b
Opérateurs arithmétiques		
+	Addition	a+b
-	Soustraction	a-b
*	Multiplication	a*b
/	Division	a/b
%	Modulo (reste)	a%b
-	Négation	-a
Opérateurs d'affectation		
++	Augmentation de 1	x++ (post fixé) ++x (préfixé)
--	Diminution de 1	x-- (post fixé) --x (préfixé)
Opérateurs logiques		
&&	ET (and)	x&& y
	OU (or)	x y
!	NON logique, négation (NOT)	!x
Opérateur conditionnel		
Expression1 ? Expression2 : Expression3		

- Les opérateurs sont classifiés en 3 types:
- Unaire: qui admettent un unique opérande. E.g. $-x$
 - Binaire: qui possèdent 2 operands. E.g. $2+3$
 - Ternaire: il existe un unique opérateur ternaire qui traite 3 opérandes, c'est l'opérateur conditionnel

Il existe des opérateurs de bits qui exécutent des opérations logiques ET, OU, OUX, NON

et des opérations de décalage sur tous les bits, pris un à un, de leurs opérandes entiers.

III. Opérateurs

Les règles des priorités en C sont nombreuses et complexes, le tableau suivant montre les priorités des opérateurs (les plus utilisés) par ordre décroissant:

Description	Opérateurs
Parenthèses	() []
Opérateur de cast	(type de données)
Opérateur de taille	sizeof
Opérateur d'adresse	&
Opérateur d'indirection	*
Opérateur de négation	- !
Incrément, décrétement	++ --
Opérateurs arithmétiques	* / % + -
Opérateurs de comparaison	> >= < <= == !=
Opérateurs logiques	&&
Opérateur conditionnel	? :
Opérateurs d'affectation	= += -= *= /= % >>= <<= &= = ^=
Opérateur séquentiel	,

Exemples:

```
a = b + ( c = 3 ) ;
```

Cette expression affecte à c la valeur 3, puis affecte à a la valeur b + c.

```
a - b - c;
```

Cette expression se décompose si on veut ajouter des parenthèses en : (a - b) - c et certainement pas: a - (b - c)

```
a = b = c ;
```

se décompose en b = c suivi de a = b.

L'opérateur unaire **cast** permet de convertir le type d'une donnée en un autre type.

Ainsi, l'expression **(double) x** transforme le type d'une variable **int** en **double**.

IV. Traitements conditionnels et boucles

Traitements conditionnels:

On appelle traitement conditionnel un portion de code qui n'est pas exécutée systématiquement, c.à.d des instructions dont l'exécution est conditionnée par le succès d'un test.

1. if: L'instruction ne s'exécute que si la condition est vraie.

La syntaxe est :

```
if (<expression>) {  
    <instruction>;  
}
```

Exemples :

```
if (i>5 && i>0)  
    printf(" i est positif et plus grand que 5. \n") ;  
printf(" je ne fais pas parti de l'instruction if.\n") ;
```

```
if (5)  
    printf("la condition est VRAIE.\n") ;
```

```
if (0)  
    printf("cette phrase n'est jamais affichée. \n") ;
```

```
#include <stdio.h>  
int main ( )  
{  
    int i ;  
    printf ( " Saisissez une valeur: " ) ;  
    scanf ( "%d" , &i ) ;  
    if ( i == 0)  
    {  
        printf ( "Vous avez saisi une valeur  
nulle \n . " ) ;  
    }  
    printf ("Adios!");  
    return 0 ;  
}
```

Notez bien qu'il n'y a pas de point-virgule après la parenthèse de if


IV. Traitements conditionnels et boucles

2. **If ... else:** Dans certains cas, on pourra faire exécuter des instructions si la condition est vraie ou si elle est fausse.

```
if (<expression>) {  
    <instruction> ;  
}  
else {  
    <instruction>  
}
```

Les instructions délimitées par **if** et ses accolades sont exécutées si le test est vérifié, et les instructions délimitées par **else** et ses accolades sont exécutées si le test n'est pas vérifié.

```
#include <stdio.h>  
int main ( )  
{  
    int i ;  
    printf ( " Saisissez une valeur: " ) ;  
    scanf ( "%d" , &i ) ;  
    if ( i == 0)  
    {  
        printf ( "Vous avez saisi une valeur  
nulle \n . " ) ;  
    }  
    else  
    {  
        printf( "La valeur que vous avez saisi,  
a savoir %d , n'est pas nulle. \n" , i  
        ) ;  
    }  
    return 0 ;  
}
```

 Notez la présence de l'opérateur de comparaison ==. N'utilisez jamais = pour comparer deux valeurs !.

IV. Traitements conditionnels et boucles

3. **switch**: Pour éviter l'imbrication de plusieurs instructions « **if** » quand il y a plus de 2 choix, on préférera utiliser l'instruction de choix multiples « **switch** ».

La syntaxe est la suivante :

```
switch (<expression>
{
    case constante_1 : <instruction_1> ; break;
    case constante_2 : <instruction_2> ; break;
    ...
    case constante_N : <instruction_N> ; break;
    default      :      [<instruction(s)> ;
}
```

```
switch (numeroMois)
{
    case 1 : printf ("janvier") ; break ;
    case 2 : printf ("février") ; break ;
    case 3 : printf ("mars") ; break ;
    case 4 : printf ("avril") ; break ;
    case 5 : printf ("mai") ; break ;
    case 6 : printf ("juin") ; break ;
    case 7 : printf ("juillet") ; break ;
    case 8 : printf ("aout") ; break ;
    case 9 : printf ("septembre") ; break ;
    case 10 : printf ("octobre") ; break ;
    case 11 : printf ("novembre") ; break ;
    case 12 : printf ("decembre") ; break ;
    default : printf ( "Je ne connais pas ce
mois ... " ) ;
}
```

N'oubliez surtout
pas les **break** pour
sortir du **switch** !

IV. Traitements conditionnels et boucles

Boucles:

Une boucle permet d'exécuter plusieurs fois de suite une même séquence d'instructions. Cet ensemble d'instructions s'appelle le corps de la boucle.

Chaque exécution du corps d'une boucle s'appelle une itération, ou plus informellement un passage dans la boucle.

Lorsque l'on s'apprête à exécuter la première itération, on dit que l'on rentre dans la boucle.

Lorsque la dernière itération est terminée, on dit qu'on sort de la boucle.
Il existe trois types de boucle :

```
while  
do ... while  
for
```

IV. Traitements conditionnels et boucles

1. **while**: la boucle **while**, étant une boucle, est réalisée autant de fois que la condition reste VRAIE.

La syntaxe est la suivante :

```
while(<condition>)  
{  
    <instructions>  
}
```

NE PAS OUBLIER:
- l'initialisation de la boucle.
- la fin de la condition dans la liste d'instructions à exécuter afin de ne pas obtenir une boucle infinie !

2. **do... while**: la boucle **do...while** teste sa condition après exécution de l'instruction du corps de la boucle. Une boucle **do ... while** est donc exécutée donc au moins une fois.

La syntaxe est la suivante :

```
do  
{  
    <instruction(s)> ;  
    ... ;  
}  
while(<expression>) ;
```

Un des usages les plus courant de la boucle **do...while** est le contrôle de saisie.

```
#include<stdio.h>  
int main ( )  
{  
    int i = 1 ; /* initialisation*/  
    while ( i <= 5)  
    {  
        printf ( "%d " , i ) ;  
        i++;  
    }  
    return 0 ;  
}
```

Affiche:
1 2 3 4 5

```
#include <stdio.h>  
int main ( )  
{  
    int i = 1 ;  
    do  
    {  
        printf ( "%d " , i ) ;  
        i++;  
    }  
    while ( i <= 5 ) ;  
    return 0 ;  
}
```

Affiche:
1 2 3 4 5

IV. Traitements conditionnels et boucles

3. **for:** **for** est une boucle qui teste une condition avant d'exécuter les instructions qui en dépendent. Les instructions sont exécutées tant que la condition remplie est VRAIE.

La syntaxe est la suivante :

```
for(<initialisation> ; <condition> ; <pas>)  
{  
    <instructions>  
}
```


L'<initialisation> est une instruction exécutée avant le premier passage dans la boucle.

La <condition> est évaluée avant chaque passage dans la boucle, si elle n'est pas vérifiée, on ne passe pas dans la boucle et l'exécution de la boucle est terminée.

La <pas> est une instruction exécutée après chaque passage dans la boucle.

```
#include<stdio.h>  
int main ( )  
{  
    int i ;  
    for (i = 1 ; i <= 5 ;i++){  
        printf ( "%d " , i ) ;  
    }  
    return 0 ;  
}
```

Affiche:
1 2 3 4 5



On utilise une boucle **for** lorsque l'on connaît en entrant dans la boucle combien d'itérations devront être faites.
Par exemple, n'utilisez pas une boucle **for** contrôler une saisie !

IV. Traitements conditionnels et boucles: TD 2

1. Ecrire un programme qui teste si la valeur saisie par l'utilisateur est nulle ou non en l'affichant sur l'écran.
2. Ecrire un programme qui donne le signe du produit des deux variables i et j ($i \times j$) saisies par l'utilisateur sans les multiplier.
3. Ecrire un programme qui calcul le carré des N premiers nombres, où N est saisi par l'utilisateur. Utilisez la boucle **while**, puis **do...while** puis **for**.
4. Calculer la somme S des N nombres impaires Pour N entre 1 et 150. Utilisez la boucle **for**.
5. Ecrire un programme qui calcule et affiche proprement la table de multiplication.

while répète le code tant que la condition est vraie

La boucle do...while est exécutée au moins une fois

Chaque programme a besoin d'une fonction main()

Les boucles for représentent une manière plus compacte pour écrire les boucles

#include importe des bibliothèques pour des trucs comme les entrées sorties

if exécute le code si la condition est vraie

(float) 5 transforme 5 en 5.0

L'opérateur de comparaison est ==. N'utilisez jamais = pour comparer deux valeurs!