

## Cours 3

Programmation impérative

Emna Chebbi, Revekka Kyriakoglou

# Plan du cours

## 1 Rappel

## 2 Iteration

- Boucle while
- Boucle for
- do-while

## 3 Instructions break, continue et goto

- break et continue
- goto

# Rappel

- Le point-virgule ( ; ) est un **terminateur d'instruction** !



( En Pascal est un séparateur.)

- Les accolades { } sont utilisés pour regrouper des déclarations et des instructions afin de obtenir une **instruction composée (bloc)**.
- On peut déclarer des variables dans n'importe quel bloc.
- L'accolade fermante qui termine un bloc n'est pas suivi d'un point virgule.

1

## 1

3/20

# L'instruction while



Dans la boucle **while**, le programme répète un bloc d'instructions tant qu'une certaine condition est vraie.

```
while ( condition )  
    instruction
```

- Tout d'abord, l'expression est évaluée.
- Si l'expression est non nulle, l'instruction s'exécute et l'expression est évaluée de nouveau. Ensuite, la procédure recommence (c'est-à-dire que l'expression est évaluée)
- Si l'expression est nulle, la procédure se termine et nous sortons de la boucle while.



La condition s'appelle une **condition d'arrêt**.



## Exercise 1

Essayez de décrire ce qui se passe avec le code suivant :

```
expression 1;  
while ( expression 2; ) {  
    instruction  
    expression 3;  
}  
expression 4;
```

- Quelle est l'expression de comparaison ?
- Peut-on supprimer les points-virgules ; ?
- Ce qui arrivera si nous supprimons l'expression 3 ?

# Exemple (puissance)

## Exemple

*La fonction suivant calcule la puissance d'un nombre entier et retourne comme résultat un nombre entier.*

*(Nous ne voulons pas utiliser la fonction pow() de math.h car elle fonctionne avec des flottants et cela peut poser des problèmes d'arrondi).*

```
/*calculer la puissance d'un nombre entier*/
int power (int n, int k){
    int i = 0 ; //initialisation
    int result = 1; //initialisation      1 (produit)
    while (i < k ){
        result = result * n;
        i = i + 1; //progression
    }
    return result;
}
```

# L'instruction for

L'instruction for (= pour)

```
for ( expression 1; expression 2; expression 3){  
    instruction  
}
```

équivalent à :

```
expression 1;  
while (expression 2;) {  
    instruction  
    expression 3;  
}
```



Les trois parties d'une boucle for dont des expressions.



```
for (expression 1; expression 2; expression 3){  
    instruction  
}
```

En général :

- Les expressions 1 et 3 sont des affectations ou des appels de fonctions.
- L'expression 2 est une expression de comparaison.



Les expressions sont toutes les trois facultatives, **mais** les points-virgules doivent toujours être présents.



S'il manque les expressions 1 ou 3, ces expressions disparaissent simplement du développement du `for`.



S'il manque l'expression 2, on considère qu'elle est toujours vraie, qui crée une **boucle infinie**.

## ? Exercise 2

■ Quel est l'affichage du programme suivant ? Pourquoi ?

```
#include<stdio.h>
int main(void){
    int i = 0;
    for (printf("new\n"); i < 3 && printf("test\n"); i++, printf("iteration\n")){
        printf("block\n");
    }
    return 0;
}
```

# Virgule dans le for



La virgule "," est souvent utilisé dans les instructions for, comme vous pouvez le voir dans l'exemple suivant :

```
#include <stdio.h>
int main (void){
    int sumk ;
    int k;
    for(sumk = 1, k = 1; k < 10; k++, sumk += k)
        printf("k=%d_et_sum=%d\n", k, sumk);
    return 0;
}
```

## ? Exercise 3

■ Quel est l'affichage du programme suivant ? Pourquoi ?

```
#include <stdio.h>
int main( void ){
    int i;
    for (i = 0; i < 30; i++){
        switch(i){
            case 0: i += 5;
            case 1: i += 2;
            case 5: i += 3;
            default: i += 4;
        }
        printf("%d\n",i);
    }
    return 0;
}
```

# Exemple for

```
#include <stdio.h>
/* affiche la table Fahrenheit-Celsius*/
int main( void ){
    int fahr;
    float celcius;
    for(fahr = 0; fahr <= 300; fahr = fahr +20){
        celcius = (5.0/9.0)*(fahr-32);
        printf("%3d└─>└%.1f\n", fahr, celcius);
    }
    return 0;
}
```

# L'instruction do-while

L'instruction do-while :

```
do{  
    instruction  
}  
while (expression);
```



L'instruction s'exécute, puis l'expression est évaluée.

- Si l'expression est vraie, on évalue de nouveau l'instruction, et ainsi de suite.
- Si l'expression est fausse, la boucle se termine.

# while vs do-while

```
//while
int i = 0;
while(i > 0){
    printf("%d",i);
    i--;
}
```

```
//do-while
int i = 0;
do{
    printf("%d",i);
    i--;
}
while(i > 0);
```

**Output : No Output**

**Output : 0;**



while et for testent leur condition d'arrêt **en tête de boucle**.



do-while teste sa condition **à la fin**, après chaque passage dans le corps de la boucle.



do-while s'exécute **au moins une fois** !



## Exercise 4



Comment le programme suivant pourrait-il être écrit avec do-while ?  
Lequel choisiriez-vous dans ce cas (do-while ou while) ?

```
//while
int n;
printf("Entrer_un_entier:");
scanf("%d", &n);

while(n < 5){
    printf("Entrer_un_entier:");
    scanf("%d",&n);
}
printf("plus_grand_que_5");
```



# break et continue



## break



L'instruction **break** permet de sortir directement d'une boucle `for`, `while` ou `do`, de même que pour `switch`.



## continue



L'instruction **continue** relance immédiatement la boucle `for`, `while` ou `do`, dans laquelle se trouve.



L'instruction `continue` **ne s'applique pas** aux instructions `switch`.



Si on place l'instruction `continue` dans un `switch` qui se trouve à l'intérieur d'une boucle, on se passe directement à l'itération suivante de la boucle.



## Exercise 5

■ Quel est l'affichage du programme suivant ? Pourquoi ?

```
#include<stdio.h>
int main(void){
    int i = -10;
    while (i <= 5){
        printf("%d\n",i);
        if(i >= 0){
            break;
        }
        else{
            i++;
            continue;
        }
        printf("Programmation_imperative");
    }
    return 0;
}
```

# Instruction goto



## goto

L'instruction goto nous permet de transférer le contrôle du programme à l'étiquette "label" spécifiée. Le label est un identifiant. Lorsque l'instruction goto est rencontrée, le programme saute à label : et commence à exécuter le code.

```
goto label;
```

```
...
```

```
...
```

```
label:
```

```
statement;
```

```
statement1;
```

```
if(condition)
```

```
    goto label;
```

```
statement2;
```

```
statement3;
```

```
statement4;
```

```
label:
```

```
statement5;
```

The `goto` statement breaks the normal flow of execution in the program and takes the control to statement5, without executing the statements 3 and 4.

goto statement

# Exemple

```
#include <stdio.h>
int main( void ){
    int sum=0;
    for(int i = 0; i<=10; i++){
        sum = sum+i;
        if(i==5){
            goto addition;
        }
    }
    addition:
    printf("%d", sum);
    return 0;
}
```

## ? Exercise 6

Quel est le sum qui va être imprimé ?

Pouvez-vous réécrire le programme sans utiliser goto ?



La condition dans laquelle l'utilisation de goto est préférable est lorsque nous devons interrompre les multiples boucles en même temps.

```
#include <stdio.h>
int main( void ){
    int i, j, k;
    for(i=0; i<5; i++){
        for(j=0; j<5; j++){
            for(k=0; k<3; k++){
                printf("d_d_d\n", i, j, k);
                if(j == 3){
                    goto out;
                }
            }
        }
    }
    out:
    printf("came_out_of_the_loop");
    return 0;
}
```