

# Cours 4

Les boucles `do while`

Le type `bool`

L'instruction `switch`

Vincent Lepetit

`vincent.lepetit@epfl.ch`

Les boucles `for` que nous avons vues la semaine dernière permettent de répéter une partie du programme.

Les boucles `for` sont utilisées quand le nombre de répétitions est connu *avant* d'entrer dans la boucle.

Selon le problème à résoudre, il arrive qu'on ne connaisse pas combien de fois la boucle devra être exécutée.

On utilise alors une boucle `while`.

# Un premier exemple

Supposons qu'on veuille écrire un programme qui tire un nombre au hasard et demande à l'utilisateur de le deviner.

Si on ne laisse qu'un seul essai à l'utilisateur, le programme peut s'écrire ainsi:

```
int nombre_a_deviner = 1 + rand() % 10;
int nombre_entre;

cout << "Entrez un nombre entre 1 et 10" << endl;
cin >> nombre_entre;

if (nombre_entre == nombre_a_deviner)
    cout << "Trouve" << endl;
else
    cout << "Perdu, le nombre etait " << nombre_a_deviner << endl;
```

# Un premier exemple

On veut maintenant laisser à l'utilisateur autant d'essais qu'il faut pour trouver le nombre.

Il faut utiliser une boucle.

On ne peut pas utiliser de boucle `for`, puisqu'on veut répéter la boucle tant que l'utilisateur n'a pas trouvé, et on ne sait pas donc pas combien de fois il faudra répéter la boucle (mais voir la remarque du transparent suivant).

Dans ce type de cas, on utilise une boucle `while`:

```
int nombre_a_deviner = 1 + rand() % 10;
int nombre_entre;

do {
    cout << "Entrez un nombre entre 1 et 10" << endl;
    cin >> nombre_entre;
} while(nombre_entre != nombre_a_deviner);

cout << "Trouve" << endl;
```

Indique le début de la boucle `while`

`do {`

```
cout << "Entrez un nombre entre 1 et 10" << endl;  
cin >> nombre_entre;
```

```
} while(nombre_entre != nombre_a_deviner);
```

Corps de la boucle;  
Il est exécuté au moins une fois.

Condition. Elle est testée juste après chaque exécution du corps de la boucle:

- si elle est vraie, le corps de la boucle est exécuté une nouvelle fois;
- si elle est fausse, on sort de la boucle.

## Remarque:

*On ne peut pas utiliser de boucle `for`, puisqu'on veut répéter la boucle tant que l'utilisateur n'a pas trouvé, et on ne sait pas donc pas combien de fois il faudra répéter la boucle.*

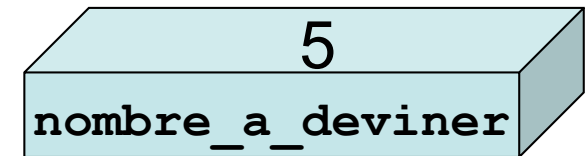
Ce n'est pas tout à fait vrai. En C et en C++, on peut utiliser une boucle `for` pour faire ce que ferait une boucle `while`, mais c'est uniquement parce que la forme de la boucle `for` est très générale. Les autres langages de programmation disposent également de l'équivalent de la boucle `for` du C et C++, mais cet équivalent est souvent plus restrictif.

Néanmoins, le programme est plus simple à écrire et à lire si on utilise

- une boucle `for` quand le nombre de répétition est connu avant d'entrer dans la boucle,
- une boucle `while` sinon.

# Pas-à-pas:

## Supposons que le nombre à deviner, tiré par l'ordinateur, est 5



```
int nombre_a_deviner = 1 + rand() % 10;  
int nombre_entre;
```

```
do {  
    cout << "Entrez un nombre entre 1 et 10" << endl;  
    cin >> nombre_entre;  
} while(nombre_entre != nombre_a_deviner);
```

```
cout << "Trouve" << endl;
```

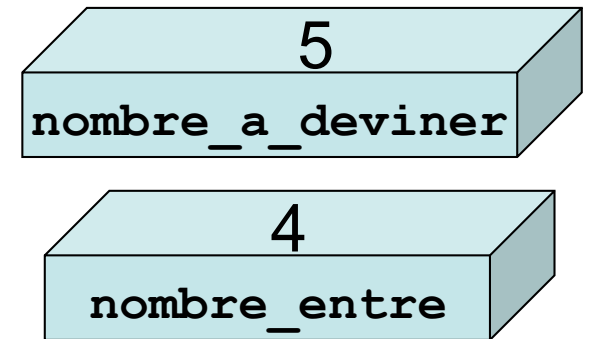
Comme la condition est vraie (rappel: != veut dire différent), on continue la boucle...

# ... et que l'utilisateur entre 4

```
int nombre_a_deviner = 1 + rand() % 10;
int nombre_entre;

do {
    cout << "Entrez un nombre entre 1 et 10" << endl;
    cin >> nombre_entre;
    } while(nombre_entre != nombre_a_deviner);

cout << "Trouve" << endl;
```



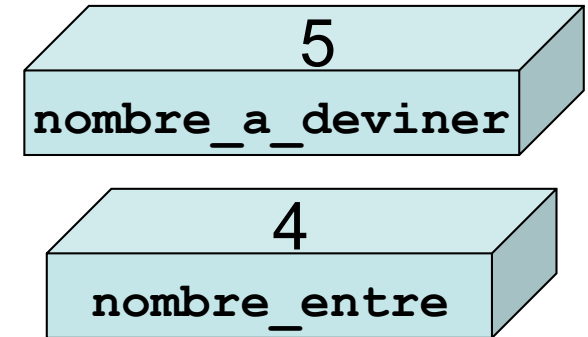
# La condition est testée

Comme elle est vraie (rappel: `!=` veut dire différent),  
on continue la boucle: le corps de la boucle va être ré-exécuté:

```
int nombre_a_deviner = 1 + rand() % 10;
int nombre_entre;

do {
    cout << "Entrez un nombre entre 1 et 10" << endl;
    cin >> nombre_entre;
    → while (nombre_entre != nombre_a_deviner);

    cout << "Trouve" << endl;
}
```



# Le corps de la boucle est donc ré-exécuté

```
int nombre_a_deviner = 1 + rand() % 10;  
int nombre_entre;
```

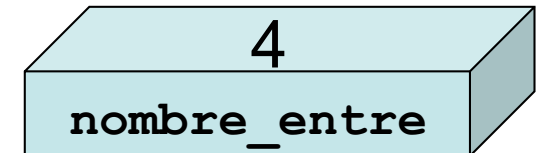
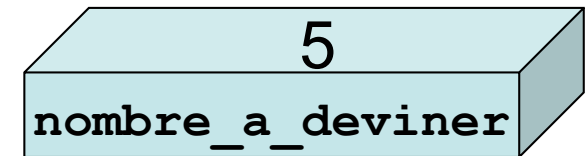
```
do {
```

```
    cout << "Entrez un nombre entre 1 et 10" << endl;
```

```
    cin >> nombre_entre;
```

```
    } while(nombre_entre != nombre_a_deviner);
```

```
cout << "Trouve" << endl;
```

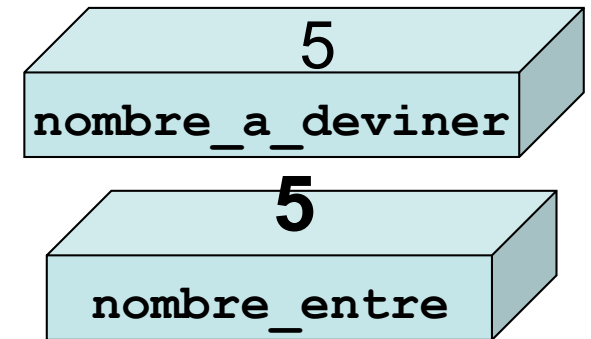


# Supposons que cette fois, l'utilisateur entre 5

```
int nombre_a_deviner = 1 + rand() % 10;
int nombre_entre;

do {
    cout << "Entrez un nombre entre 1 et 10" << endl;
    cin >> nombre_entre;
    } while(nombre_entre != nombre_a_deviner);

cout << "Trouve" << endl;
```

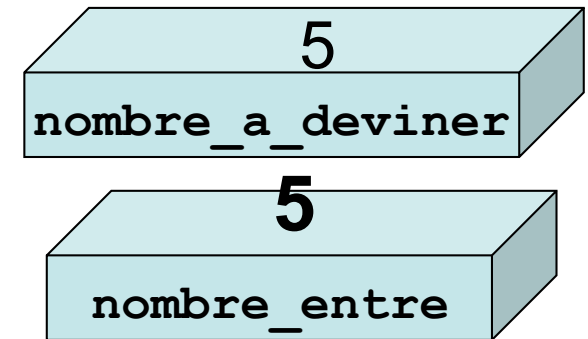


# Cette fois, la condition est fausse, et la boucle s'arrête

```
int nombre_a_deviner = 1 + rand() % 10;
int nombre_entre;


do {
    cout << "Entrez un nombre entre 1 et 10" << endl;
    cin >> nombre_entre;
    → while (nombre_entre != nombre_a_deviner);

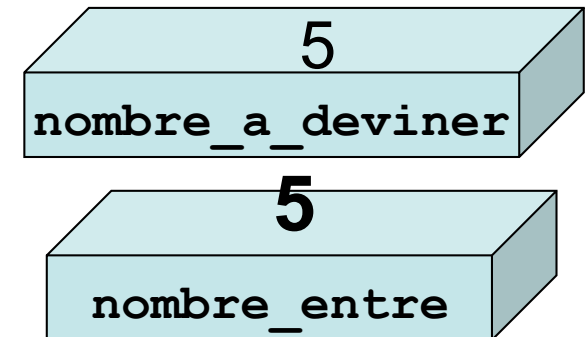
    cout << "Trouve" << endl;
```



# et les instructions suivant la boucle sont exécutées

```
int nombre_a_deviner = 1 + rand() % 10;  
int nombre_entre;  
  
do {  
    cout << "Entrez un nombre entre 1 et 10" << endl;  
    cin >> nombre_entre;  
} while(nombre_entre != nombre_a_deviner);
```

 cout << "Trouve" << endl;



# Syntaxe de l'instruction `do...while`

La boucle `while` en C et C++ a deux variantes. Celle que nous venons de voir est l'instruction `do...while`, qui a donc la forme suivante:

```
do
    instruction;
while (condition);
```

```
do
    { bloc }
while (condition);
```

## Remarques:

- Comme pour l'instruction `if`:
  - La condition peut utiliser des opérateurs logiques.
  - Les parenthèses autour de la condition sont obligatoires.
- Il y a un point-virgule après la condition. // *N'y a **PAS** de point-virgule après le `do`.*
- Les instructions à l'intérieur de la boucle `do...while` sont toujours exécutées au moins une fois.
- Si la condition ne devient jamais fausse, les instructions dans la boucle sont répétées indéfiniment ! Il faut taper Ctrl-C pour arrêter le programme.

# Syntaxe de l'instruction `while...`

L'autre forme est l'instruction `while...` qui a la structure suivante:

```
while (condition)
instruction;
```

```
while (condition)
{ bloc }
```

## Différences entre la forme `do...while` et la forme `while...`:

- Contrairement à la forme `do...while`, la condition est ici testée ***avant*** d'entrer dans la boucle:  
Si la condition est fausse, les instructions dans la boucle ne sont donc pas exécutées;
- Il n'y a **PAS** de point-virgule après la condition.

# Erreurs classiques

Rappel: Il n'y a pas de ; à la fin de l'instruction `for`:

```
for(i = 0; i < 10; i++); // !!  
    cout << i << endl;
```

Le point-virgule est considéré comme l'instruction nulle, qui sera ici répétée 10 fois.

Il n'y a pas non plus de ; à la fin du `while...`

```
while(i < 10); // !!  
    i++;
```

Dans ce cas, le point-virgule est considéré comme le corps de la boucle, et l'instruction `i++` est après la boucle. Si `i` est inférieur à 10, on entre dans la boucle pour ne jamais en ressortir puisque la valeur de `i` ne sera jamais modifiée.

En revanche, il y a un point-virgule à la fin du `do..while`:

```
do  
    i++;  
while(i < 10);
```

# Lien entre do...while **et** while...

```
do
  instruction;
while (condition);
```

**est équivalent à:**

```
instruction;
while (condition)
  instruction;
```

```
while (condition)
  instruction;
```

**est équivalent à:**

```
do
  if (condition)
    instruction;
while (condition);
```

# Quand utiliser la boucle `while` ?

## Quand utiliser la boucle `for` ?

Quand le nombre d'itérations (de répétitions) est connu avant d'entrer dans la boucle, utiliser `for`

Sinon, utiliser `while`:

– quand les instructions doivent être effectuées au moins une fois, utiliser `do...while`:

```
do
{
    instructions;
} while (test);
```

–Sinon, utiliser la forme `while...`

```
while (test)
{
    instructions;
}
```

Problèmes classiques faisant intervenir  
des boucles `for` ou `while`

Différents problèmes reviennent à calculer les termes d'une suite par récurrence de la forme:

$$U_0 = \textit{valeur initiale}$$

et

$$U_n = f(U_{n-1}) \text{ ou}$$

$$U_n = f(U_{n-1}, n)$$

$U_0 = \textit{valeur initiale}$

et

$U_n = f(U_{n-1})$  ou  $U_n = f(U_{n-1}, n)$

Par exemple, si on veut simplement calculer les N premières valeurs de  $(U_n)$ , le code sera alors de la forme:

```
type U = valeur initiale;  
for(int n = 1; n <= N; n++)  
{  
    ...  
    U = f(U, n);  
    ...  
}
```

`type` vaut `float` si `U` doit contenir des nombres à virgule,  
`int` si `U` doit contenir des valeurs entières.

```
type U = valeur initiale;  
for(int n = 1; n <= N; n++)
```

```
{
```

```
...
```

```
U =  $f(U, n)$  ;
```

```
..
```

```
}
```

La forme exacte de  $f$  dépend du problème à résoudre.

Quand cette expression est évaluée, la variable `U`  
contient la valeur de  $U_{n-1}$ .

Une fois l'affectation est exécutée, la variable `U` contient  
la valeur de  $U_n$ .

# Premier exemple: Somme des N premiers entiers

Calculer la somme des entiers de 1 à N (il y a une formule donnant directement la valeur de la somme, mais nous allons utiliser ici une boucle `for`).

Pour cela, on peut utiliser la suite:

$$U_0 = 0$$

$$U_n = U_{n-1} + n$$

On peut vérifier que cette suite calcule bien la somme recherchée:

$$U_0 = 0$$

$$U_1 = 0 + 1 = 1$$

$$U_2 = 1 + 2 = 3$$

$$U_3 = 3 + 3 = 6$$

$$U_4 = 6 + 4 = 10$$

$U_4$  contient bien la somme des 4 premiers entiers  $1 + 2 + 3 + 4$ .

# Premier exemple

La suite ( $U_n$ ) est ici:

$$U_0 = 0$$

$$U_n = U_{n-1} + n$$

```
type U = valeur initiale;  
for(int n = 1; n <= N; n++)  
{  
    U = f(U, n);  
}
```

Cette ligne devient:

```
int U = 0;
```

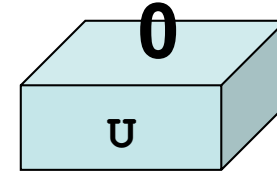
Cette ligne devient:

```
U = U + n;
```

Le code final est donc:

```
int U = 0;  
for(int n = 1; n <= N; n++)  
    U = U + n;
```

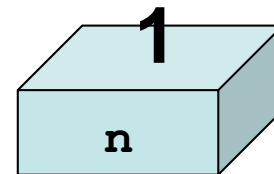
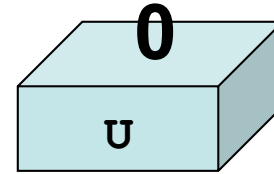
# Pas-à-pas



```
→ int U = 0;  
   for(int n = 1; n <= N; n++)  
       U = U + n;
```

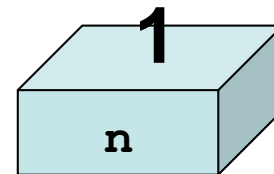
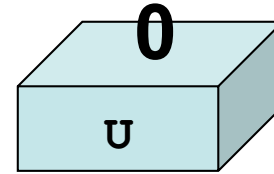
# Pas-à-pas

```
int U = 0;  
→ for(int n = 1; n <= N; n++)  
    U = U + n;
```

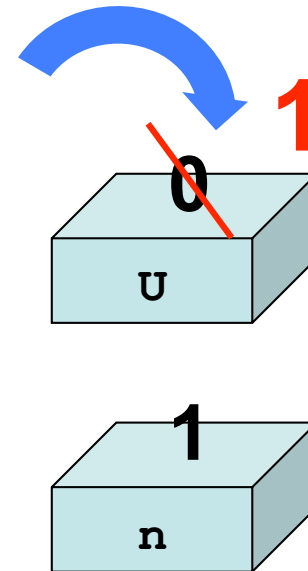


# Pas-à-pas

```
int U = 0;  
for(int n = 1; n <= N; n++)  
→   U = U + n;
```

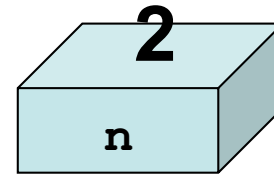
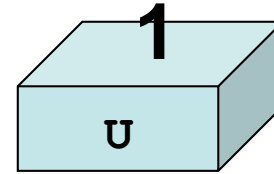


$U + n$  vaut  $0 + 1$ , qui vaut 1

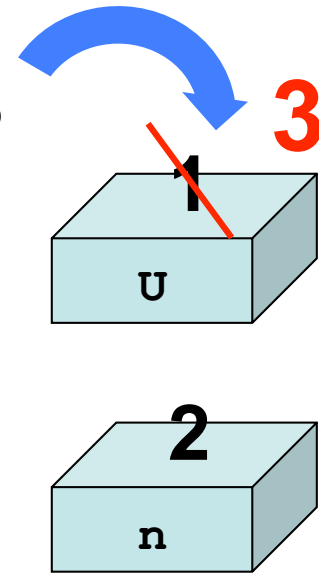


```
int U = 0;  
for(int n = 1; n <= N; n++)  
→   U = U + n;
```

```
int U = 0;  
→ for(int n = 1; n <= N; n++)  
    U = U + n;
```



$U + n$  vaut  $1 + 2$ , qui vaut  $3$



```
int U = 0;  
for(int n = 1; n <= N; n++)  
→   U = U + n;
```

# Variante 1: Somme de nombres entrés par l'utilisateur

On veut calculer la somme de N valeurs entrées par l'utilisateur:

Le programme précédent:

```
int U = 0;
for(int n = 1; n <= N; n++)
    U = U + n;
```

devient:

```
int somme = 0;
for(int n = 1; n <= N; n++)
{
    int valeur;
    cout << "Entrez une valeur:" << endl;
    cin >> valeur;
    somme = somme + valeur;
}
```

(somme est ici un meilleur nom de variable que U)

# Pas-à-pas

On veut calculer la somme de N valeurs entrées par l'utilisateur:

Le programme précédent:

```
int U = 0;
for(int n = 1; n <= N; n++)
    U = U + n;
```

devient:

```
int somme = 0;
for(int n = 1; n <= N; n++)
{
    int valeur;
    cout << "Entrez une valeur:" << endl;
    cin >> valeur;
    somme = somme + valeur;
}
```

somme est ici un meilleur nom de variable que U.

# Variante 2: Moyenne de valeurs

On veut calculer la *moyenne* de N valeurs entrées par l'utilisateur:

Il suffit d'ajouter une ligne après la boucle du programme précédent:

```
int somme = 0;
for(int n = 1; n <= N; n++)
{
    int valeur;
    cout << "Entrez une valeur:" << endl;
    cin >> valeur;
    somme = somme + valeur;
}
float moyenne = float(somme) / N;
```

moyenne est de type `float`, car elle contiendra un nombre à virgule.

Le *cast* `float(somme)` permet d'éviter d'utiliser la division entière.

# Deuxième exemple: Produit des N premiers entiers

Calculer le produit des entiers de 1 à N. (Ce produit s'appelle factoriel N, et s'écrit N!)

Pour cela, on peut utiliser la suite:

$$U_0 = 1$$

$$U_n = U_{n-1} \times n$$

On peut vérifier que cette suite calcule bien la valeur recherchée:

$$U_0 = 1$$

$$U_1 = 1 \times 1 = 1$$

$$U_2 = 1 \times 2 = 2$$

$$U_3 = 2 \times 3 = 6$$

$$U_4 = 6 \times 4 = 24$$

$U_4$  contient bien le produit des 4 premiers entiers  $1 \times 2 \times 3 \times 4$ .

# Deuxième exemple

La suite ( $U_n$ ) est ici:

$$U_0 = 1$$

$$U_n = U_{n-1} \times n$$

```
type U = valeur initiale;  
for(int n = 1; n <= N; n++)  
{  
    U = f(U, n);  
}
```

Cette ligne devient:

```
int U = 1;
```

Cette ligne devient:

```
U = U * n;
```

Le code final est donc:

```
int U = 1;  
for(int n = 1; n <= N; n++)  
    U = U * n;
```

# Troisième exemple: Déterminer la valeur d'un minimum

On veut trouver la valeur minimum de  $\cos(n)$  pour  $n$  compris entre 1 et  $N$ .

La suite  $(U_n)$  est ici:

$$U_0 = \cos(1)$$

$$\text{Si } \cos(n) < U_{n-1} \text{ alors } U_n = \cos(n)$$

$$\text{Si } \cos(n) \geq U_{n-1} \text{ alors } U_n = U_{n-1}$$

Le code est donc:

```
float U = cos(1);
for(int n = 1; n <= N; n++)
    if (cos(n) < U)
        U = cos(n);
    else
        U = U; // cette ligne ne fait rien et peut etre supprimee
```

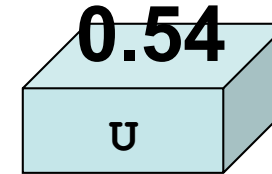
# Troisième exemple: Déterminer la valeur d'un minimum

```
float U = cos(1);  
for(int n = 1; n <= N; n++)  
    if (cos(n) < U)  
        U = cos(n);  
    else  
        U = U; // cette ligne ne fait rien et peut etre supprimee
```

Le code final est donc:

```
float U = cos(1);  
for(int n = 1; n <= N; n++)  
    if (cos(n) < U)  
        U = cos(n);
```

# Pas-à-pas



```
→ float U = cos(1);  
   for(int n = 1; n <= N; n++)  
       if (cos(n) < U)  
           U = cos(n);
```

cos(1) vaut 0.54...

cos(2) vaut -0.41...

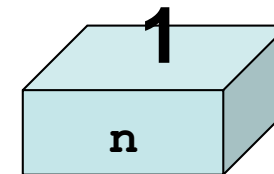
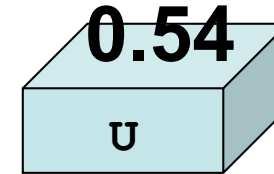
cos(3) vaut -0.98...

cos(4) vaut -0.65...

cos(5) vaut 0.28...

# Pas-à-pas

```
float U = cos(1);  
→ for (int n = 1; n <= N; n++)  
    if (cos(n) < U)  
        U = cos(n);
```



cos(1) vaut 0.54...  
cos(2) vaut -0.41...  
cos(3) vaut -0.98...  
cos(4) vaut -0.65...  
cos(5) vaut 0.28...

# Pas-à-pas

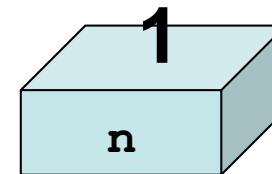
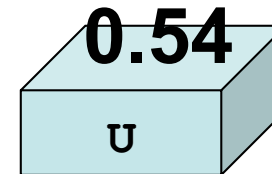
```
float U = cos(1);  
for(int n = 1; n <= N; n++)
```



```
if (cos(n) < U)
```

```
    U = cos(n);
```

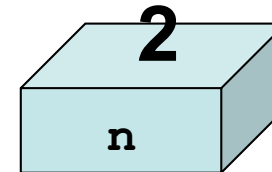
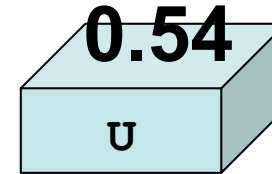
la condition est fausse



cos(1) vaut 0.54...  
cos(2) vaut -0.41...  
cos(3) vaut -0.98...  
cos(4) vaut -0.65...  
cos(5) vaut 0.28...

# Pas-à-pas

```
float U = cos(1);  
→ for(int n = 1; n <= N; n++)  
    if (cos(n) < U)  
        U = cos(n);
```



cos(1) vaut 0.54...  
cos(2) vaut -0.41...  
cos(3) vaut -0.98...  
cos(4) vaut -0.65...  
cos(5) vaut 0.28...

# Pas-à-pas

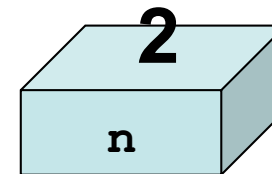
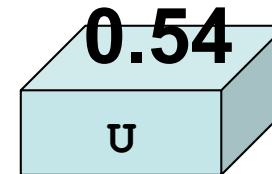
```
float U = cos(1);  
for(int n = 1; n <= N; n++)
```



```
if (cos(n) < U)
```

```
    U = cos(n);
```

la condition est vraie



cos(1) vaut 0.54...

**cos(2) vaut -0.41...**

cos(3) vaut -0.98...

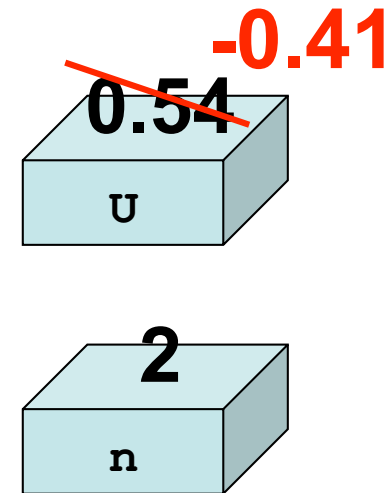
cos(4) vaut -0.65...

cos(5) vaut 0.28...

# Pas-à-pas

```
float U = cos(1);  
for(int n = 1; n <= N; n++)  
    if (cos(n) < U)  
        → U = cos(n);
```

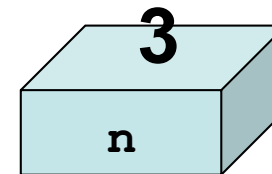
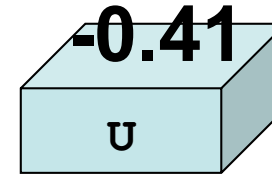
cos(1) vaut 0.54...  
**cos(2) vaut -0.41...**  
cos(3) vaut -0.98...  
cos(4) vaut -0.65...  
cos(5) vaut 0.28...



# Pas-à-pas

```
float U = cos(1);  
→ for(int n = 1; n <= N; n++)  
    if (cos(n) < U)  
        U = cos(n);
```

cos(1) vaut 0.54...  
cos(2) vaut -0.41...  
cos(3) vaut -0.98...  
cos(4) vaut -0.65...  
cos(5) vaut 0.28...



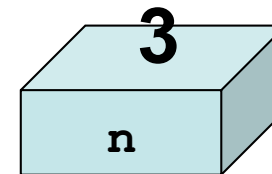
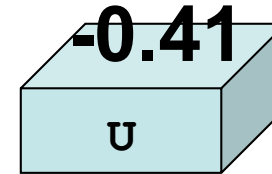
# Pas-à-pas

```
float U = cos(1);  
for(int n = 1; n <= N; n++)
```



```
if cos(n) < U
```

```
U = cos(n); la condition est vraie
```



cos(1) vaut 0.54...

cos(2) vaut -0.41...

**cos(3) vaut -0.98...**

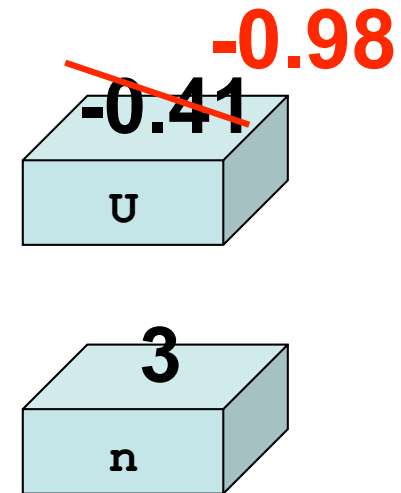
cos(4) vaut -0.65...

cos(5) vaut 0.28...

# Pas-à-pas

```
float U = cos(1);  
for(int n = 1; n <= N; n++)  
    if (cos(n) < U)  
        → U = cos(n);
```

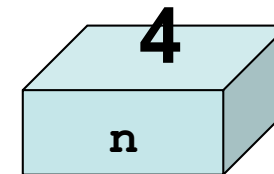
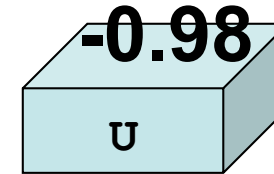
cos(1) vaut 0.54...  
cos(2) vaut -0.41...  
**cos(3) vaut -0.98...**  
cos(4) vaut -0.65...  
cos(5) vaut 0.28...



# Pas-à-pas

```
float U = cos(1);  
for(int n = 1; n <= N; n++)  
→ if (cos(n) < U)  
    U = cos(n);
```

la condition est fausse



cos(1) vaut 0.54...  
cos(2) vaut -0.41...  
cos(3) vaut -0.98...  
**cos(4) vaut -0.65...**  
cos(5) vaut 0.28...

# Pas-à-pas

```
float U = cos(1);  
→ for(int n = 1; n <= N; n++)  
    if (cos(n) < U)  
        U = cos(n);
```

**-0.98**  
U

**5**  
n

cos(1) vaut 0.54...  
cos(2) vaut -0.41...  
cos(3) vaut -0.98...  
cos(4) vaut -0.65...  
cos(5) vaut 0.28...

# Variante: Trouver le rang du minimum

On ajoute une variable pour mémoriser le rang du minimum:

```
float U = cos(1);  
int rang = 1;  
for(int n = 1; n <= N; n++)  
    if (cos(n) < U)  
    {  
        U = cos(n);  
        rang = n;  
    }
```

Considérons maintenant le cas où le nombre d'itérations n'est pas connu à l'avance mais dépend de la valeur de  $U_n$ .

Le code est maintenant de la forme:

```
type U = valeur initiale;  
int n = 1;  
do {  
    ...  
    U = f(U, n);  
    ...  
    n++; // equivalent à n = n + 1;  
} while(condition(U));
```

# Exemple

Calculer les premiers termes de la suite

$$U_0 = 1$$

$$U_n = \frac{U_{n-1}}{n}$$

tant que  $U_n > 0.0001$ .

```
float U = 1;  
int n = 1;  
do {  
    U = U / n;  
    n++;  
} while(U > 0.0001);
```

# Variante

Calculer la *somme* des premiers termes de la suite

$$U_0 = 1$$

$$U_n = \frac{U_{n-1}}{n}$$

tant que  $U_n > 0.0001$ .

On peut utiliser la suite  $V_n$  avec:

$$V_0 = 0$$

$$V_n = V_{n-1} + U_n$$

```
float U = 1, V = 0;  
int n = 1;  
do {  
    U = U / n;  
    V = V + U;  
    n++;  
} while(U > 0.0001);
```

# Retour sur l'exemple du début du cours

Supposons qu'on veuille indiquer à l'utilisateur le nombre d'essais qu'il lui a fallu pour trouver le nombre.

Il suffit d'ajouter une variable initialisée à 0, et incrémentée de 1 à chaque tour de boucle:

```
int nombre_a_deviner = 1 + rand() % 10;
int nombre_entre;
int nombre_essais = 0;

do {
    cout << "Entrez un nombre entre 1 et 10" << endl;
    cin >> nombre_entre;
    nombre_essais++;
} while(nombre_entre != nombre_a_deviner);

cout << "Trouve" << endl;
cout << "Il vous a fallu " << nombre_essais << "essai(s)." << endl;
```

Trouver la condition d'une boucle  
`while`

# Nombre d'essais limité

On veut maintenant ne laisser que 3 essais à l'utilisateur. Il faut changer la condition:

```
int nombre_a_deviner = 1 + rand() % 10;
int nombre_entre;
int nombre_essais = 0;

do {
    cout << "Entrez un nombre entre 1 et 10" << endl;
    cin >> nombre_entre;
    nombre_essais++;
} while(nombre_entre != nombre_a_deviner ... );
```

Comment modifier la condition pour que la boucle s'arrête quand le nombre d'essais dépasse 3 ?

Rappel: la boucle s'arrête quand la condition devient fausse.

# Nombre d'essais limité

```
int nombre_a_deviner = 1 + rand() % 10;
int nombre_entre;
int nombre_essais = 0;

do {
    nombre_essais++;
    cout << "Entrez un nombre entre 1 et 10" << endl;
    cin >> nombre_entre;
} while(nombre_entre != nombre_a_deviner && nombre_essais < 3);

if (nombre_entre == nombre_a_deviner)
    cout << "Trouve" << endl;
else
    cout << "Perdu. Le nombre etait " << nombre_a_deviner << endl;
```

# Comment trouver la condition ?

Il n'est pas toujours facile de trouver l'expression correcte de la condition pour la boucle `while`.

Le "truc" est d'essayer d'exprimer ce test en langage naturel. Par exemple:

On veut répéter la boucle

*tant que* le nombre entré est différent du nombre à deviner, **et**  
qu'il reste des essais. Il reste des essais si le nombre est inférieur à 3.

D'où:

```
while (nombre_entre != nombre_a_deviner && nombre_essais < 3);
```

Dérouler ensuite à la main le programme pas-à-pas pour vérifier qu'il est correct.

C'est pour ça qu'il est important de donner aux variables des noms qui correspondent à ce qu'elles représentent. Le code est plus facile à écrire et à comprendre.

# Suite de l'exemple

Attention au test:

```
if (nombre_entre == nombre_a_deviner)  
    cout << "Trouve" << endl;  
else  
    cout << "Perdu. Le nombre etait " << nombre_a_deviner << endl;
```

Si on avait écrit:

```
if (nombre_essais < 3)  
    cout << "Trouve" << endl;  
else  
    cout << "Perdu. Le nombre etait " << nombre_a_deviner << endl;
```

le programme afficherait "Perdu. ..." quand l'utilisateur trouve au troisième essai.

# Forcer l'utilisateur à entrer un nombre supérieur ou égal à 1

Mettons les lignes:

```
cout << "Entrez un nombre entre 1 et 10" << endl;  
cin >> nombre_entre;
```

dans une nouvelle boucle `while`:

```
do {  
    do {  
        cout << "Entrez un nombre entre 1 et 10" << endl;  
        cin >> nombre_entre;  
    } while ( ??? );  
    nombre_essais++;  
} while (nombre_entre
```

Quelle condition utiliser pour que la boucle se répète si l'utilisateur entre un nombre inférieur à 1 ?

Rappel: la boucle se répète tant que la condition est vraie.

# Forcer l'utilisateur à entrer un nombre supérieur ou égal à 1

```
do {  
    do {  
        cout << "Entrez un nombre  
        cin >> nombre entre;  
    } while (nombre_entre < 1);  
    nombre_essais++;  
} while(nombre_entre != nombre_a_deviner && nombre_essais < 3);
```

## La condition

nombre\_entre < 1  
est vraie si l'utilisateur entre un  
nombre inférieur à 1.

# Forcer l'utilisateur à entrer un nombre entre 1 et 10

```
do {  
    do {  
        cout << "Entrez  
        cin >> nombre_e  
    } while ( ??? );  
    nombre_essais++;  
} while(nombre_entr
```

La nouvelle condition doit être vraie si le nombre n'est pas valide.

Le nombre n'est pas valide si il est inférieur à 1 **ou** si il est supérieur à 10.

# Forcer l'utilisateur à entrer un nombre entre 1 et 10

## La condition

`nombre_entre < 1 || nombre_entre > 10`  
est vraie si le nombre n'est pas valide.

La boucle se répète alors.

```
do {  
    do {  
        cout << "La boucle se répète alors."  
        cin >> nombre_entre;  
    } while (nombre_entre < 1 || nombre_entre > 10);  
    nombre_essais++;  
} while (nombre_entre != nombre_a_deviner && nombre_essais < 3);
```

# VERIFIER que la condition est correcte

VERIFIER que la condition marche pour un certain nombre de valeurs:

```
while(nombre_entre < 1 || nombre_entre > 10);
```

Si `nombre_entre` vaut 5, la condition est fausse, et la boucle va s'arrêter. C'est bien ce qu'on veut puisque 5 est une valeur valide.

Si `nombre_entre` vaut 0, la condition est vraie, et la boucle va continuer. C'est bien ce qu'on veut puisque 0 n'est pas une valeur valide.

1. Ecrire un programme qui demande à l'utilisateur d'entrer des notes. L'utilisateur indiquera qu'il veut arrêter en entrant la valeur -1. Le programme affichera alors la moyenne des notes.
2. Modifier le programme pour limiter le nombre de notes à 10. L'utilisateur peut toujours entrer -1 s'il souhaite arrêter.
3. Modifier le programme pour qu'il affiche la note la plus grande.
4. Modifier le programme pour qu'il affiche le nombre de notes supérieures à 4.

```
float somme = 0;
...
do {
    float note;
    cout << "Entrez une note" << endl;
    cin >> note;
    ...
} while(...);
...
```

**Le type** `bool`

# Le type `bool`

Le type `bool` (pour boolean, ou booléen) est le type des conditions.

Il peut donc prendre deux valeurs: vrai ou faux, qui sont définies par les constantes `true` et `false`.

On peut déclarer des variables de type `bool`:

```
bool cond1, cond2;
```

```
cond1 = true;
```

```
cond2 = false;
```

Les opérateurs `&&` `||` et `!` peuvent donc être utilisés entre des expressions de type `bool`:

```
if (cond1 && cond2)
{
    ...
}
```

# Utilisation

```
if (cond1 == true)
```

```
    a = b;
```

**est correct mais il vaut mieux écrire**

```
if (cond1)
```

```
    a = b;
```

**En effet:**

- **quand cond1 est vraie: (cond1 == true) est vraie**
- **quand cond1 est fausse: (cond1 == true) est fausse**

**les deux tests**

```
    cond1 == true
```

**et**

```
    cond1
```

**sont donc équivalents.**

# Utilisation

De même:

```
if (cond2 == false)
    a = b;
```

est correct mais il vaut mieux écrire

```
if (!cond2)
    a = b;
```

En effet:

- **Quand cond2 est vraie:** (cond2 == false) est fausse;
- **quand cond2 est fausse:** (cond2 == false) est vraie.

Et également:

```
cond1 = (cond2 == true);
est correct mais il vaut mieux écrire
cond1 = cond2;
```

Voir également l'exemple suivant.

# Utiliser un booléen

Le test `nombre_entre != nombre_a_deviner` se répète deux fois dans notre exemple précédent. On peut éviter cette répétition en stockant la valeur dans un booléen:

```
// ...
bool trouve;

do {
    nombre_essais++;
    cout << "Entrez un nombre entre 1 et 10" << endl;
    cin >> nombre_entre;
    trouve = nombre_entre == nombre_a_deviner;
} while(!trouve && nombre_essais < 3);
if (trouve)
    cout << "Trouve" << endl;
else
    //...
```

**Rappel:** Le symbole `!` est le non logique. La condition peut donc se lire donc:  
*tant que le nombre n'est pas trouvé et qu'il reste des essais,*  
on continue la boucle.

**L'instruction** switch

On veut écrire un programme qui demande à l'utilisateur d'entrer le rayon d'un cercle, puis qui lui demande s'il souhaite en obtenir le diamètre, le périmètre ou l'aire:

- Si l'utilisateur entre 1, le diamètre sera affiché.
- Si l'utilisateur entre 2, le périmètre sera affiché.
- Si l'utilisateur entre 3, l'aire sera affichée.

```
int main(int argc, char ** argv)
{
    float rayon, diametre, perimetre, aire;
    int choix;

    cout << "Entrez le rayon" << endl;
    cin >> rayon;

    cout << "Tapez 1 pour obtenir le diametre" << endl;
    cout << "Tapez 2 pour obtenir le perimetre" << endl;
    cout << "Tapez 3 pour obtenir l aire" << endl;
    cin >> choix;

    // ...
}
```

# Solution

```
if (choix == 1)
{
    diametre = 2 * rayon; // D = 2 R
    cout << "Le diametre vaut " << diametre << endl;
}
else if (choix == 2)
{
    perimetre = 2 * 3.14159 * rayon; // P = 2 Pi R
    cout << "Le perimetre vaut " << perimetre << endl;
}
else if (choix == 3)
{
    aire = 3.14159 * rayon * rayon; // A = Pi R2
    cout << "L aire vaut " << aire << endl;
}
```

Sans les instructions `else`, le programme aurait le même comportement, puisque une seule des 3 conditions est vraie à la fois.

Néanmoins, ils aident le lecteur du programme à comprendre qu'un seul calcul est effectué.

# L'instruction `switch`

Le programme précédent exécutait des actions différentes, en fonction de la valeur de la variable `choix`.

Cela était réalisé à l'aide d'une cascade de `if`, testant tous des valeurs différentes pour `choix`:

```
if (choix == 1)
{
    diametre = 2 * rayon; // D = 2 R
    cout << "Le diametre vaut " << diametre << endl;
}
else if (choix == 2)
{
    perimetre = 2 * 3.14159 * rayon; // P = 2 Pi R
    cout << "Le perimetre vaut " << perimetre << endl;
}
else if (choix == 3)
{
    aire = 3.14159 * rayon * rayon; // A = Pi R2
    cout << "L aire vaut " << aire << endl;
}
```

# L'instruction `switch`

Dans un tel cas, on peut utiliser l'instruction `switch` pour remplacer les `if` successifs:

```
switch(choix)
```

```
{
```

```
  case 1:
```

```
    diametre = 2 * rayon; // D = 2 R
```

```
    cout << "Le diametre vaut " << diametre << endl;
```

```
    break;
```

```
  case 2:
```

```
    perimetre = 2 * 3.14159 * rayon; // P = 2 Pi R
```

```
    cout << "Le perimetre vaut " << perimetre << endl;
```

```
    break;
```

```
  case 3:
```

```
    aire = 3.14159 * rayon * rayon; // A = Pi R2
```

```
    cout << "L aire vaut " << aire << endl;
```

```
    break;
```

```
  default:
```

```
    cout << "Tapez 1, 2 ou 3" << endl;
```

```
}
```

# Syntaxe de l'instruction `switch`

```
switch (expression)  
{  
  case constante_1:  
    suite d'instructions1  
    break;  
  case constante_2:  
    suite d'instructions2  
    break;  
  case constante_3:  
    suite d'instructions3  
    break;  
  default:  
    suite d'instructions default  
}
```

Obligatoirement une expression entière  
(par exemple de type `int` ou `short`)

Obligatoirement des constantes entières

Attention à ne pas oublier le mot-clé `break` !!

La *suite d'instructions default* est exécutée si aucune des *constante\_1*, *constante\_2*... ne correspond à l'*expression*.  
Pas obligatoire.