

Cours 3

Fonctions mathématiques

L'instruction `if`

Opérateurs de comparaison et opérateurs logiques

Les boucles `for`

Règles d'écriture de programmes

Vincent Lepetit

`vincent.lepetit@epfl.ch`

Fonctions mathématiques

Fonctions mathématiques

La bibliothèque standard du C++ fournit les fonctions mathématiques usuelles. Par exemple:

```
#include <iostream>
using namespace std;
#include <cmath>
```

Il faut inclure `cmath`



```
int main(int argc, char ** argv)
{
    float angle;
    float s;

    angle = 10 * 3.14159 / 180;
    s = sin(angle);
    // ...
}
```

→ `sin(angle)` calcule le sinus de la valeur contenue dans la variable `angle`.

Fonctions mathématiques

Autres exemples d'expressions contenant des fonctions mathématiques:

```
y = 2 * cos(a) * sin(a);
```

```
b_1 = sqrt(delta);  
      (sqrt est la fonction racine carrée)
```

```
angle = atan(l / w);  
       (atan est la fonction arc tangente).
```

Fonctions mathématiques

- `sin`
- `cos`
- `tan`
- `asin` sinus inverse ou arc sinus
- `acos`
- `atan`
- `atan2` `atan2(y, x)` fournit la valeur de l'arc-tangente de y / x
- `sinh` sinus hyperbolique ou sh
- `cosh`
- `tanh`
- `exp`
- `log` logarithme népérien ou *ln*
- `log10` logarithme à base 10 ou *log*
- `pow` `pow(x, y)` fournit la valeur de x^y
- `sqrt` racine carrée
- `ceil` `ceil(x)` renvoie le plus petit entier qui ne soit pas inférieur à x : `ceil(2.6) = 3`
- `floor` `floor(x)` renvoie le plus grand entier qui ne soit pas supérieur à x : `floor(2.6) = 2`
- `fabs` valeur absolue

Nombres pseudo-aléatoires

La fonction `rand()` retourne un nombre entier (de type `int`) apparemment tiré au hasard (en fait, un nombre pseudo-aléatoire).

Ce nombre est compris entre 0 et `RAND_MAX`, qui est une valeur définie dans le fichier `stdlib.h`.

La fonction `rand()` n'a pas d'argument, les parenthèses doivent néanmoins être présentes.

Nombres pseudo-aléatoires

```
#include <iostream>
using namespace std;
#include <stdlib.h>

int main(int argc, char ** argv)
{
    cout << rand() << endl;
    cout << rand() << endl;

    return 0;
}
```

donnera par exemple:

```
1481765933
1085377743
```

Notez qu'il faut ajouter `#include <stdlib.h>` au début du programme.

Nombres pseudo-aléatoires

Comment obtenir un nombre au hasard entre 0 et 9 ?

→ il suffit de prendre le reste de la division par 10:

```
int n = rand() % 10;
```

Comment obtenir un nombre au hasard entre 1 et 10 ?

→ il suffit de prendre le reste de la division par 10, et d'ajouter 1:

```
int n = rand() % 10 + 1;
```

*Comment obtenir un nombre **flottant** au hasard entre 0 et 1 ?*

→ `rand()` est entre 0 et `RAND_MAX`, il suffit de diviser `rand()` par `RAND_MAX`.

Attention à la division !

```
float x = float(rand()) / RAND_MAX;
```

*Comment obtenir un nombre **flottant** au hasard entre -1 et 1 ?*

→ `float(rand()) / RAND_MAX`; est entre 0 et 1

donc `2 * float(rand()) / RAND_MAX`; est entre 0 et 2

donc `2 * float(rand()) / RAND_MAX - 1`; est entre -1 et 1

```
float x = 2 * float(rand()) / RAND_MAX - 1;
```

Nombres pseudo-aléatoires

La suite entière récurrente:

$$I_{n+1} = a I_n + c \text{ modulo } m$$

est composée de nombres qui *paraissent* être tirés au hasard, avec de "bonnes" valeurs pour a , c et m , quelque soit le premier terme I_0 .

Par exemple, si on prend $a = 3$, $c = 5$, $m = 17$, avec $I_0 = 8$:

$$I_1 = (3 \times I_0 + 5) \text{ modulo } 17 = 24 + 5 \text{ modulo } 17 = 29 \text{ modulo } 17 = 12$$

$$I_2 = (3 \times I_1 + 5) \text{ modulo } 17 = 7$$

$$I_3 = (3 \times 7 + 5) \text{ modulo } 17 = 9$$

$$I_4 = (3 \times 9 + 5) \text{ modulo } 17 = 15$$

...

$$I_{16} = (3 \times 1 + 5) \text{ modulo } 17 = \mathbf{8}$$

Puisque l'opération modulo par m ne peut retourner que m résultats différents, la suite générée est périodique. En pratique, il suffit de prendre m suffisamment grand pour ne pas s'en apercevoir:

La fonction `rand()` utilise une formule similaire avec $m = \text{RAND_MAX} = 2147483647$.

Tirer des nombres aléatoires

La fonction `srand()` permet de définir le premier terme de la suite.

Les instructions suivantes afficheront les mêmes valeurs à chaque exécution:

```
srand(0); // Le premier terme I0 vaut 0
cout << rand() << endl;
cout << rand() << endl;
cout << rand() << endl;
```

Pour obtenir des nombres différents à chaque exécution, il faut initialiser la suite avec une valeur différente à chaque exécution.

Un moyen pratique pour cela est d'utiliser la fonction `time()`, qui renvoie le nombre de secondes depuis le 1^{er} Janvier 1970 à 00h00 UTC:

```
srand( time(0) );
cout << rand() << endl;
cout << rand() << endl;
cout << rand() << endl;
```

Pour pouvoir utiliser la fonction `time()`, il faut ajouter au début du programme:

```
#include <time.h>
```

srand et rand

- La fonction `srand()` permet de définir le premier terme de la suite.
 - A appeler une fois, au début du programme.
 - Il faut lui donner une valeur en paramètre:
 - soit une constante entière, et le programme tirera toujours les mêmes valeurs à chaque exécution;
 - soit on utilise la fonction `time`, et les valeurs aléatoires sont différentes pour chaque exécution du programme.
- La fonction `rand` retourne un nombre aléatoire:
 - A appeler à chaque fois qu'on veut un nombre aléatoire.
 - Pas de paramètre.

Instructions de contrôle

Jusqu'ici, toutes les instructions des programmes étaient exécutées, et les unes après les autres.

Les instructions de contrôle permettent de changer ce comportement.

Dans ce cours, nous allons voir:

- l'instruction `if`, qui permet de sauter certaines parties du programme, si certaines conditions sont remplies;
- l'instruction `for`, qui permet de répéter plusieurs fois une partie du programme.

L'instruction `if`

Choix entre deux instructions

```
int main(int argc, char ** argv)
{
    int n;

    cout << "Entrez votre nombre:" << endl;
    cin >> n;

    if (n < 5)
        cout << "Votre nombre est plus petit que 5." << endl;
    else
        cout << "Votre nombre est plus grand ou egal a 5." << endl;

    cout << "Au revoir" << endl;

    return 0;
}
```

Condition

```
if (n < 5)
```

```
    cout << "Votre nombre est plus petit que 5." << endl;
```

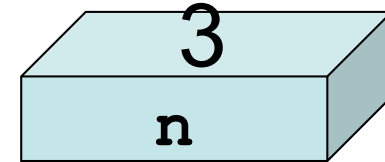
```
else
```

```
    cout << "Votre nombre est plus grand ou egal a 5." << endl;
```

Cette instruction sera exécutée si la condition est *vraie*.

Cette instruction sera exécutée si la condition est *fausse*.

Pas-à-pas



```
int n;
```

```
cout << "Entrez votre nombre:" << endl;
```

```
cin >> n;
```

```
if (n < 5)
```

```
    cout << "Votre nombre est plus petit que 5." << endl;
```

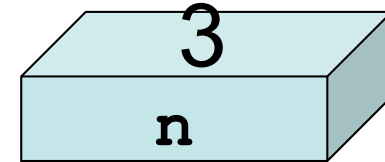
```
else
```

```
    cout << "Votre nombre est plus grand ou egal a 5." << endl;
```

```
cout << "Au revoir" << endl;
```

Supposons que l'utilisateur tape 3. `n` prend la valeur 3.

Pas-à-pas



```
int n;
```

```
cout << "Entrez votre nombre:" << endl;
```

```
cin >> n;
```

```
→ if (n < 5)
```

```
    cout << "Votre nombre est plus petit que 5." << endl;
```

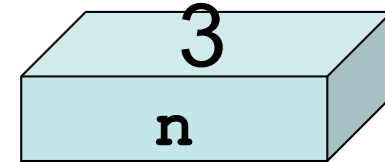
```
else
```

```
    cout << "Votre nombre est plus grand ou egal a 5." << endl;
```

```
cout << "Au revoir" << endl;
```

La condition est testée. Elle est vraie (n vaut 3, et est donc inférieur à 5).
L'ordinateur va donc exécuter la première instruction du `if`.

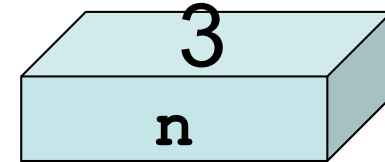
Pas-à-pas



```
int n;  
  
cout << "Entrez votre nombre:" << endl;  
cin >> n;  
  
if (n < 5)  
    cout << "Votre nombre est plus petit que 5." << endl;  
else  
    cout << "Votre nombre est plus grand ou egal a 5." << endl;  
  
cout << "Au revoir" << endl;
```

La première instruction du `if` est exécutée. Le programme affiche:
Votre nombre est plus petit que 5.

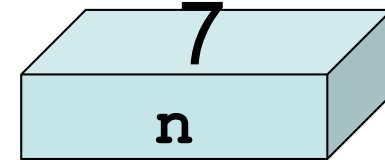
Pas-à-pas



```
int n;  
  
cout << "Entrez votre nombre:" << endl;  
cin >> n;  
  
if (n < 5)  
    cout << "Votre nombre est plus petit que 5." << endl;  
else  
    cout << "Votre nombre est plus grand ou egal a 5." << endl;  
  
→ cout << "Au revoir" << endl;
```

Le programme continue avec l'instruction placée après le `if`.

Pas-à-pas



```
int n;  
  
cout << "Entrez votre nombre:" << endl;  
cin >> n;  
  
if (n < 5)  
    cout << "Votre nombre est plus petit que 5." << endl;  
else  
→   cout << "Votre nombre est plus grand ou egal a 5." << endl;  
  
cout << "Au revoir" << endl;
```

Si l'utilisateur avait tapé 7, c'est la deuxième instruction qui aurait été exécutée:
Votre nombre est plus grand ou egal a 5.

Choix entre deux *blocs* d'instructions

Il est possible de placer plusieurs instructions dans chacune des deux parties du choix, à *condition* de les regrouper en un bloc:

Un bloc est une suite d'instructions placées entre accolades { et }

Exemple, pour déterminer le plus grand de 2 nombres:

```
if (p > n)
{
    cout << "p est plus grand que n." << endl;
    le_plus_grand = p;
}
else
{
    cout << "n est plus grand ou egal a p." << endl;
    le_plus_grand = n;
}
```

Choix entre deux *blocs* d'instructions

Les lignes

```
if (p > n)
{
    cout << "p est plus grand que n." << endl;
    le_plus_grand = p;
}
else
{
    cout << "n est plus grand ou egal a p." << endl;
    le_plus_grand = n;
}
```

suivent le schéma suivant:

```
if (condition)
    bloc1
else
    bloc2
```

Cas particulier d'instruction `if`

Il n'est pas nécessaire qu'une instruction `if` comporte une partie introduite par `else`.

Ainsi, avec les instructions suivantes:

```
if (n < p)
    cout << "n est plus petit que p" << endl;
cout << "Au revoir" << endl;
```

l'instruction

```
    cout << "n est plus petit que p" << endl;
```

est exécutée si la condition `n < p` est vraie.
Si `n` est supérieur ou égal à `p`, *l'instruction `if` ne fait rien.*

Puis, dans tous les cas, on exécute l'instruction:

```
cout << "Au revoir" << endl;
```

Ce cas particulier peut évidemment s'employer avec un bloc.

Les formes possibles de l'instruction `if`

```
if (condition)
  instruction1;
else
  instruction2;
```

```
if (condition)
  bloc1
else
  bloc2
```

```
if (condition)
  bloc1
else
  instruction2;
```

```
if (condition)
  instruction1;
else
  bloc2
```

```
if (condition)
  bloc1
```

```
if (condition)
  instruction1;
```

Les conditions

L'instruction `if` fait apparaître une **condition** entre parenthèses:

```
if (condition)  
    instruction1  
else  
    instruction2
```

Attention, la condition du `if` est toujours entourée de parenthèses.

Pour l'instant, nous n'avons rencontré que des conditions simples.

Nous allons voir maintenant comment s'écrivent les conditions d'une façon générale.

Les conditions simples

Les opérateurs de comparaison

Une **condition simple** compare deux expressions.

Elle utilise un **opérateur de comparaison**, comme < ou >.

Opérateur	Signification
==	égal
<	inférieur
>	supérieur
<=	inférieur ou égal
>=	supérieur ou égal
!=	différent

Opérateurs de comparaison du langage C

Les opérateurs de comparaison

Attention:

- Les opérateurs `==` `<=` `>=` `!=` s'écrivent avec 2 caractères. Ceux-ci ne doivent pas être séparés par des espaces.

- Il ne faut pas confondre:

l'opérateur == (qui teste l'égalité) et

l'affectation =

Comparer des float

Il est dangereux de comparer deux expressions de type `float` (ou `double`):

```
float x = 1. / 3.;  
  
if (3 * x == 1.) // !!!  
    ...
```

A cause des imprécisions de calcul, il est à peu près certain que la condition sera fausse, même si elle est vraie mathématiquement.

La solution est alors de tester si *la valeur absolue de la différence entre les 2 expressions est assez petite* :

```
if (fabs(3 * x - 1.) < 0.00001)  
    ...
```

Les opérateurs logiques

On peut relier des conditions simples par des opérateurs logiques.

Opérateurs logiques en C

Opérateur	Signification
&&	et
	ou
!	non

Exemples:

$(a < b) \ \&\& \ (c < d)$: est vraie si les deux conditions $(a < b)$ et $(c < d)$ sont toutes les deux vraies, est fausse dans le cas contraire.

$(a < b) \ || \ (c < d)$: est vraie si l'une au moins des conditions $(a < b)$ et $(c < d)$ est vraie, elle est fausse si les deux sont fausses.

$!(a < b)$: est vraie si $(a < b)$ est fausse, elle est fausse si $(a < b)$ est vraie.

Tables de vérité

ET (&&)	faux	vrai
faux	faux	faux
vrai	faux	vrai

OU ()	faux	vrai
faux	faux	vrai
vrai	vrai	vrai

NON (!)	faux	vrai
	vrai	faux

- a ET b est vrai *si et seulement si* les 2 conditions (a et b) sont vraies
- a OU b est vrai *si et seulement si* au moins l'une des deux conditions est vraie

Les opérateurs logiques: exemple

```
int main(int argc, char ** argv)
{
    int n;

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

    if ((n >= 1) && (n <= 10))
        cout << "correct" << endl;
    else
        cout << "incorrect" << endl;
```

Voici deux exemples d'exécution de ce programme:

```
Entrez un nombre entre 1 et 10 : 6
correct
```

```
Entrez un nombre entre 1 et 10 : 43
incorrect
```

Les choix imbriqués

L'instruction `if` suit donc le schéma:

```
if (condition1)
{
    ...
}
else
{
    ...
}
```

Les instructions figurant dans les blocs sont absolument quelconques. *Il peut donc s'agir d'autres instructions `if`.*

Les choix imbriqués

Les instructions figurant dans les blocs peuvent être d'autres instructions `if`. Voici un schéma illustrant une telle situation:

```
if (condition1)
{
    ... // exécuté si condition1 est vraie
    if (condition2)
    {
        ... // exécuté si condition1 et condition2 sont vraies
    }
    else
    {
        ... // exécuté si condition1 est vraie, et si
        } // condition2 est fausse
    }
else
{
    ... // exécuté si condition1 est fausse
}
... // exécuté dans tous les cas
```

Quand il y a moins d'`else` que de `if`

Dans le schéma précédent, il y avait autant de `else` que de `if`. *Mais comme une instruction `if` peut ne pas avoir de `else` associé, certaines formulations peuvent paraître ambiguës, par exemple:*

```
if (a < b)
if (b < c)
cout << "print 1" << endl;
else
cout << "print 2" << endl;
```

Comment ces instructions doivent-elles être interprétées ?

```
if (a < b)
  if (b < c)
    cout << "print 1..."
  else
    cout << "print 2..."
```

OU

```
if (a < b)
  if (b < c)
    cout << "print 1..."
else
  cout << "print 2..."
```

?

Quand il y a moins d'`else` que de `if`

Les concepteurs du langage C ont décidé de la règle suivante pour lever cette ambiguïté:

Un `else` se rapporte toujours au dernier `if` rencontré auquel un `else` n'a pas encore été attribué.

L'interprétation correcte est donc:

```
if (a < b)
    if (b < c)
        cout << "print 1..."
    else
        cout << "print 2..."
```

En cas de doute, on pourra toujours utiliser les accolades pour forcer l'interprétation:

```
if (a < b)
{
    if (b < c)
        cout << "print 1..."
    else
        cout << "print 2..."
}
```

Erreurs classiques

1. Le test d'égalité s'écrit ==, et pas =

```
if (a = 1) // !!!
```

est accepté mais ne teste pas si a vaut 1, et affecte la valeur 1 à a.

Utilisé avec `-Wall`, g++ affiche le warning suivant:

```
warning: suggest parentheses around assignment used as truth  
value
```

2. Ne pas mettre de ; à la fin de la condition:

```
if (a == 1); // !!!  
    cout << "A" << endl;
```

→ A est toujours affiché:

Le point-virgule est considéré comme l'instruction nulle. Le code précédent est compris par le compilateur comme:

```
if (a == 1)  
    ;  
    cout << "A" << endl;
```

le `cout` est donc situé après le `if`.

Aucun *warning* n'est affiché.

Erreurs classiques

Ne pas oublier les accolades, l'indentation ne suffit pas:

```
if (n < p)
    cout << "Croissant" << endl;
    max = p;
else
    cout << "Decroissant" << endl;
```

produit:

syntax error before "else"

L'instruction `max = p;` est déjà **en dehors** du `if`.

```
cout << "Entrez le premier nombre:" << endl;
cin >> n;
cout << "Entrez le deuxieme nombre:" << endl;
cin >> p;
```

```
if ((n < p) && (2 * n >= p))
    cout << "Test 1" << endl;
```

```
if ((n < p) || (2 * n > p))
    cout << "Test 2" << endl;
```

```
if ( ((n < p) && (2 * n > p)) || (n > p) )
    cout << "Test 3" << endl;
```

```
if (!(n > p))
{
    if ((2 * n <= p) && (3 * n > p))
        cout << "A" << endl;

    if (p % 2 == 0)
        cout << "B" << endl;
}
else
    if (3 * p > n)
        cout << "C" << endl;
```

Exercice

Qu'affiche ce programme quand l'utilisateur entre **10 et 5** ? **1 et 4** ?

- Pour le ET (&&): **les deux** conditions doivent être vraies;
- Pour le OU (||): **au moins l'une** des conditions doit être vraie.

La boucle `for`

La boucle `for`

Une boucle `for` permet de répéter plusieurs fois la même série d'instructions.
Par exemple, si on fait:

```
for(int i = 0; i < 5; i++)  
{  
    cout << "i = " << i << endl;  
    cout << "Bonjour" << endl;  
}
```

le programme affichera 5 fois la valeur de `i` et la chaîne "Bonjour":

```
i = 0  
Bonjour  
i = 1  
Bonjour  
i = 2  
Bonjour  
i = 3  
Bonjour  
i = 4  
Bonjour
```

Déclaration et initialisation:
n'est exécutée qu'une seule fois,
avant d'entrer dans la boucle

Condition:
testée avant l'exécution de chaque
tour de boucle. Si elle est fausse,
on sort de la boucle.

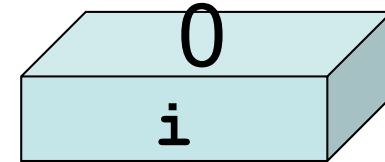
```
for (int i = 0; i < 5; i++)  
{  
    cout << "i = " << i << endl;  
    cout << "Bonjour" << endl;  
}
```

Corps de la boucle:
Bloc d'instructions qui sera
exécuté à chaque tour de boucle.

Incrémentation:
exécutée à la fin de chaque tour de boucle. Elle
permet de changer la valeur du compteur de
boucle (ici, la variable *i*).

Rappel: `i++`; ajoute 1 à la variable *i*. Cette
instruction fait la même chose que `i = i + 1`;

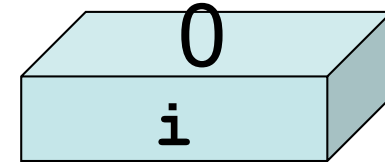
Pas-à-pas



```
→ for(int i = 0; i < 5; i++)  
  {  
    cout << "i = " << i << endl;  
    cout << "Bonjour" << endl;  
  }
```

La variable `i` est déclarée (créée) et initialisée à 0.

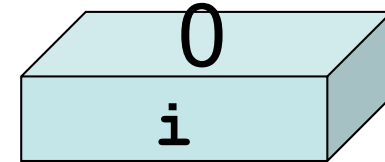
Pas-à-pas



```
→ for(int i = 0; i < 5; i++)  
  {  
    cout << "i = " << i << endl;  
    cout << "Bonjour" << endl;  
  }
```

La condition est vraie, on entre dans la boucle.

Pas-à-pas



```
for(int i = 0; i < 5; i++)  
{  
→ cout << "i = " << i << endl;  
  cout << "Bonjour" << endl;  
}
```

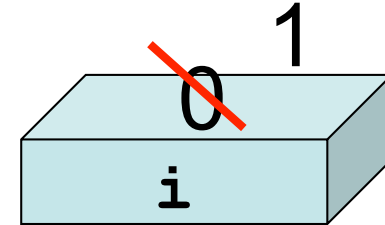
Les deux instructions dans la boucle sont exécutées, l'une après l'autre.

Le programme affiche:

i = 0

Bonjour

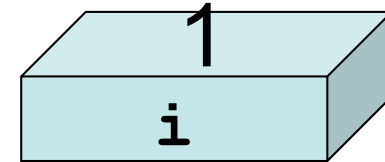
Pas-à-pas



```
→ for(int i = 0; i < 5; i++)  
{  
    cout << "i = " << i << endl;  
    cout << "Bonjour" << endl;  
}
```

`i++` est exécuté. `i` contient maintenant 1.

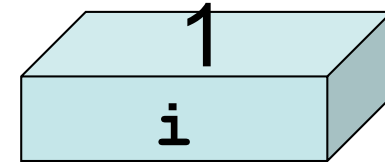
Pas-à-pas



```
→ for(int i = 0; i < 5; i++)  
  {  
    cout << "i = " << i << endl;  
    cout << "Bonjour" << endl;  
  }
```

La condition est testée à nouveau. i est toujours inférieur à 5, la boucle continue.

Pas-à-pas



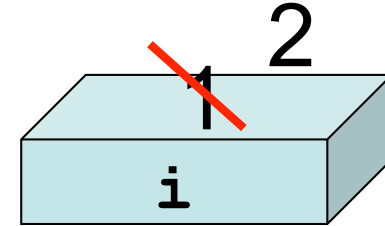
```
for(int i = 0; i < 5; i++)  
{  
→ cout << "i = " << i << endl;  
  cout << "Bonjour" << endl;  
}
```

Les deux instructions dans la boucle sont exécutées, l'une après l'autre.

Le programme affiche cette fois:

```
i = 1  
Bonjour
```

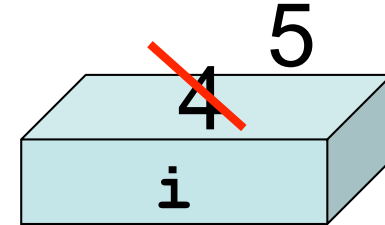
Pas-à-pas



```
→ for(int i = 0; i < 5; i++)  
  {  
    cout << "i = " << i << endl;  
    cout << "Bonjour" << endl;  
  }
```

etc...

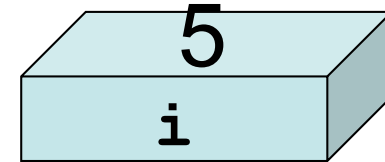
Pas-à-pas



```
→ for(int i = 0; i < 5; i++)  
  {  
    cout << "i = " << i << endl;  
    cout << "Bonjour" << endl;  
  }
```

La variable `i` finit par atteindre la valeur 5.

Pas-à-pas



```
→ for(int i = 0; i < 5; i++)  
  {  
    cout << "i = " << i << endl;  
    cout << "Bonjour" << endl;  
  }
```

La condition est testée.

Cette fois-ci, elle est fausse: `i` vaut 5 et n'est plus strictement inférieur à 5.

On sort donc de la boucle.

Pas-à-pas

```
for(int i = 0; i < 5; i++)  
{  
    cout << "i = " << i << endl;  
    cout << "Bonjour" << endl;  
}
```



Le programme continue en exécutant les instructions après la boucle.

A noter que la variable `i` "disparaît". Elle n'est déclarée que pour l'intérieur de la boucle. Elle ne peut pas être utilisée à l'extérieur de la boucle.

Affichage d'une table de multiplication

Dans le programme suivant, la même ligne ou presque est répétée 10 fois:
Une constante prend les valeurs de 1 à 10.

→ on peut utiliser une boucle `for` pour éviter cette répétition.

```
cout << "Table de multiplication par 5:" << endl;
```

```
cout << "5 multiplie par 1 vaut " << 5 * 1 << endl;
```

```
cout << "5 multiplie par 2 vaut " << 5 * 2 << endl;
```

```
cout << "5 multiplie par 3 vaut " << 5 * 3 << endl;
```

```
cout << "5 multiplie par 4 vaut " << 5 * 4 << endl;
```

```
cout << "5 multiplie par 5 vaut " << 5 * 5 << endl;
```

```
...
```

Affichage d'une table de multiplication

On peut remplacer :

```
cout << "5 multiplie par 1 vaut " << 5 * 1 << endl;  
cout << "5 multiplie par 2 vaut " << 5 * 2 << endl;  
cout << "5 multiplie par 3 vaut " << 5 * 3 << endl;  
cout << "5 multiplie par 4 vaut " << 5 * 4 << endl;  
cout << "5 multiplie par 5 vaut " << 5 * 5 << endl;  
...
```

par:

```
for(int i = 1; i <= 10; i++)  
    cout << "5 multiplie par " << i << " vaut " << 5 * i << endl;
```

La variable `i` prend successivement les valeurs de 1 à 10.

Exemples d'autres formes

```
for(int i = 0; i < 10; i++)
```

...

la variable `i` prendra les valeurs de 0 à 9 (rappel: `i++` est équivalent à `i = i + 1`);

```
for(int i = 1; i <= 10; i++)
```

...

la variable `i` prendra les valeurs de 1 à 10;

```
for(int p = 0; p < 10; p += 2)
```

...

la variable `p` prendra les valeurs de 0, 2, 4, 6, 8 (`p += 2` est équivalent à `p = p + 2`);

```
for(int k = 10; k > 0; k--)
```

...

la variable `k` prendra les valeurs 10, 9, 8 ... jusqu'à 1;

```
for(int i = 0; i >= 0; i++)
```

...

la condition étant toujours vraie, la boucle est répétée indéfiniment et la variable `i` prendra toutes les valeurs positives que le type `int` peut représenter.

La boucle `for` peut ne pas s'arrêter...

Si une boucle ne s'arrête pas, taper Ctrl-C au clavier pour stopper le programme.

Plusieurs causes sont possibles:

1. La condition est toujours vraie:

Par exemple:

```
for(i = 0; i > -1; i++) // !!!
```

2. L'incrémentation de la variable compteur est incorrecte:

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

`j` est incrémenté au lieu de `i`, `i` garde donc toujours la valeur 0, et la boucle ne s'arrête pas.

Pas de point-virgule (;) à la fin de l'instruction `for`

Les instructions suivantes n'affichent qu'une seule fois la chaîne "bonjour":

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

Le point-virgule seul est considéré comme une instruction (qui ne fait rien).

Le corps de la boucle est donc constitué de cette instruction qui ne fait rien.

`i` prendra les valeurs de 0 à 10, puis l'ordinateur sortira de la boucle, et exécutera l'instruction

```
    cout << "bonjour" << endl;
```

une seule fois.

Attention aux accolades

```
for(int i = 0; i < 5; i++)  
    cout << "i = " << i << endl;  
    cout << "Bonjour" << endl;
```

affiche:

```
i = 0  
i = 1  
i = 2  
i = 3  
i = 4  
Bonjour
```

L'instruction

```
cout << "i = " << i << endl;
```

est dans la boucle. L'instruction

```
cout << "Bonjour" << endl;
```

n'y est pas !

Ne *jamais* modifier une variable compteur à l'intérieur d'une boucle `for`

```
for(i = 0; i < 10; i++)  
{  
    ...  
    if (...)  
        i--; // !!!  
}
```

1. Ça ne fera sans doute pas ce que vous voulez: n'oubliez pas que la boucle `for`, de son côté, incrémente la variable `i`.
2. Un relecteur risque de ne pas s'apercevoir que la variable est modifiée également à l'intérieur de la boucle, et de ne pas comprendre le fonctionnement.

Qu'affichent les programmes suivants ?

A:

```
for(int i = 0; i < 5; i++)  
    cout << i * i << endl;
```

B:

```
for(int j = 0; j < 5; j++)  
    cout << j * j << ", ";  
cout << endl;
```

C:

```
for(int i = 0; i < 5; i++)  
{  
    cout << i * i << endl;  
    if (i * i % 2 == 0)  
        cout << "*" << endl;  
}
```

Moyenne de 4 notes

Sans boucle `for`, en utilisant 5 variables:

```
float note1, note2, note3, note4;
float somme;

cout << "Entrez la note numero 1" << endl;
cin >> note1;

cout << "Entrez la note numero 2" << endl;
cin >> note2;

cout << "Entrez la note numero 3" << endl;
cin >> note3;

cout << "Entrez la note numero 4" << endl;
cin >> note4;

somme = note1 + note2 + note3 + note4;
cout << "Moyenne = " << somme / 4 << endl;
```

Sans boucle `for`, en n'utilisant que 2 variables:

```
float note, somme;

cout << "Entrez la note numero 1" << endl;
cin >> note;
somme = note;

cout << "Entrez la note numero 2" << endl;
cin >> note;
somme = somme + note;

cout << "Entrez la note numero 3" << endl;
cin >> note;
somme = somme + note;

cout << "Entrez la note numero 4" << endl;
cin >> note;
somme = somme + note;

cout << "Moyenne = " << somme / 4 << endl;
```

Pour vérifier le programme précédent, supposons que l'utilisateur entre les notes 1, 2, 3 et 4:

```
float note, somme;
```

```
cout << "Entrez la note numero 1" << endl;
```

```
→ cin >> note;
```

```
somme = note;
```

```
cout << "Entrez la note numero 2" << endl;
```

```
cin >> note;
```

```
somme = somme + note;
```

```
cout << "Entrez la note numero 3" << endl;
```

```
cin >> note;
```

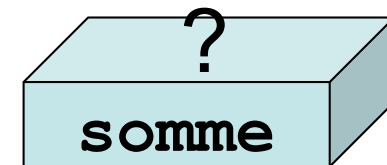
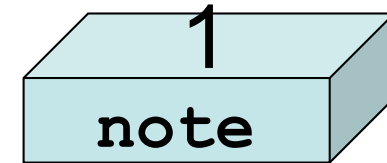
```
somme = somme + note;
```

```
cout << "Entrez la note numero 4" << endl;
```

```
cin >> note;
```

```
somme = somme + note;
```

```
cout << "Moyenne = " << somme / 4 << endl;
```



Pour vérifier le programme précédent, supposons que l'utilisateur entre les notes 1, 2, 3 et 4:

```
float note, somme;
```

```
cout << "Entrez la note numero 1" << endl;
```

```
cin >> note;
```

```
→ somme = note;
```

```
cout << "Entrez la note numero 2" << endl;
```

```
cin >> note;
```

```
somme = somme + note;
```

```
cout << "Entrez la note numero 3" << endl;
```

```
cin >> note;
```

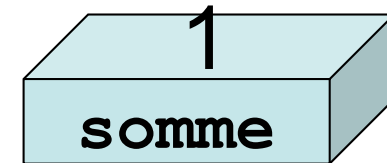
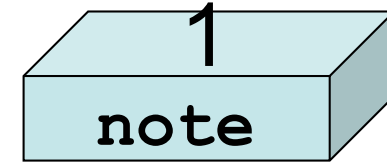
```
somme = somme + note;
```

```
cout << "Entrez la note numero 4" << endl;
```

```
cin >> note;
```

```
somme = somme + note;
```

```
cout << "Moyenne = " << somme / 4 << endl;
```



Pour vérifier le programme précédent, supposons que l'utilisateur entre les notes 1, 2, 3 et 4:

```
float note, somme;
```

```
cout << "Entrez la note numero 1" << endl;
```

```
cin >> note;
```

 `somme = note;`

```
cout << "Entrez la note numero 2" << endl;
```

```
cin >> note;
```

```
somme = somme + note;
```

```
cout << "Entrez la note numero 3" << endl;
```

```
cin >> note;
```

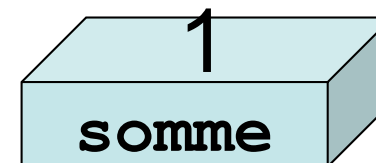
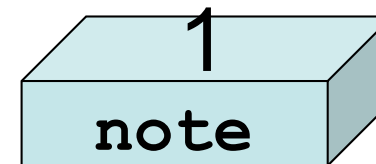
```
somme = somme + note;
```

```
cout << "Entrez la note numero 4" << endl;
```

```
cin >> note;
```

```
somme = somme + note;
```

```
cout << "Moyenne = " << somme / 4 << endl;
```



Pour vérifier le programme précédent, supposons que l'utilisateur entre les notes 1, 2, 3 et 4:

```
float note, somme;
```

```
cout << "Entrez la note numero 1" << endl;
```

```
cin >> note;
```

```
somme = note;
```

```
cout << "Entrez la note numero 2" << endl;
```

```
→ cin >> note;
```

```
somme = somme + note;
```

```
cout << "Entrez la note numero 3" << endl;
```

```
cin >> note;
```

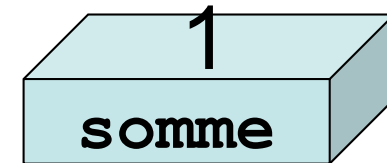
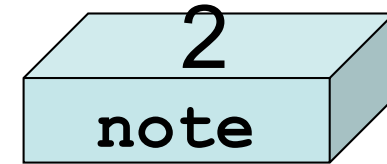
```
somme = somme + note;
```

```
cout << "Entrez la note numero 4" << endl;
```

```
cin >> note;
```

```
somme = somme + note;
```

```
cout << "Moyenne = " << somme / 4 << endl;
```



Pour vérifier le programme précédent, supposons que l'utilisateur entre les notes 1, 2, 3 et 4:

```
float note, somme;
```

```
cout << "Entrez la note numero 1" << endl;
```

```
cin >> note;
```

```
somme = note;
```

```
cout << "Entrez la note numero 2" << endl;
```

```
cin >> note;
```

```
→ somme = somme + note;
```

```
cout << "Entrez la note numero 3" << endl;
```

```
cin >> note;
```

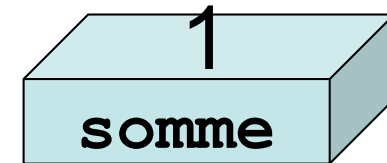
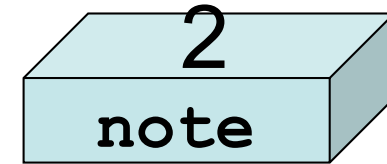
```
somme = somme + note;
```

```
cout << "Entrez la note numero 4" << endl;
```

```
cin >> note;
```

```
somme = somme + note;
```

```
cout << "Moyenne = " << somme / 4 << endl;
```



Pour vérifier le programme précédent, supposons que l'utilisateur entre les notes 1, 2, 3 et 4:

```
float note, somme;
```

```
cout << "Entrez la note numero 1" << endl;
```

```
cin >> note;
```

```
somme = note;
```

```
cout << "Entrez la note numero 2" << endl;
```

```
cin >> note;
```

```
→ somme = somme + note;
```

```
cout << "Entrez la note numero 3" << endl;
```

```
cin >> note;
```

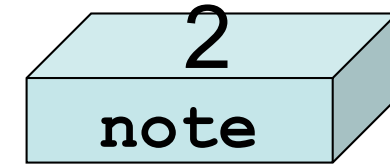
```
somme = somme + note;
```

```
cout << "Entrez la note numero 4" << endl;
```

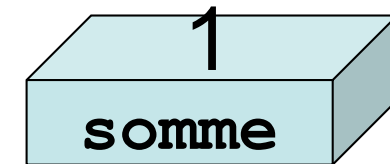
```
cin >> note;
```

```
somme = somme + note;
```

```
cout << "Moyenne = " << somme / 4 << endl;
```



$$1+2=3$$



Pour vérifier le programme précédent, supposons que l'utilisateur entre les notes 1, 2, 3 et 4:

```
float note, somme;
```

```
cout << "Entrez la note numero 1" << endl;
```

```
cin >> note;
```

```
somme = note;
```

```
cout << "Entrez la note numero 2" << endl;
```

```
cin >> note;
```

```
→ somme = somme + note;
```

```
cout << "Entrez la note numero 3" << endl;
```

```
cin >> note;
```

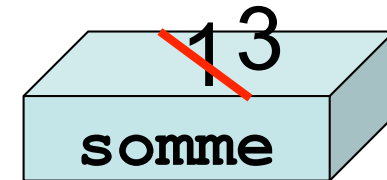
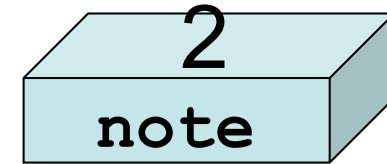
```
somme = somme + note;
```

```
cout << "Entrez la note numero 4" << endl;
```

```
cin >> note;
```

```
somme = somme + note;
```

```
cout << "Moyenne = " << somme / 4 << endl;
```



Pour vérifier le programme précédent, supposons que l'utilisateur entre les notes 1, 2, 3 et 4:

```
float note, somme;
```

```
cout << "Entrez la note numero 1" << endl;
```

```
cin >> note;
```

```
somme = note;
```

```
cout << "Entrez la note numero 2" << endl;
```

```
cin >> note;
```

```
somme = somme + note;
```

```
cout << "Entrez la note numero 3" << endl;
```

```
→ cin >> note;
```

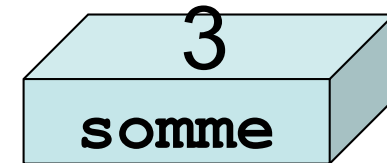
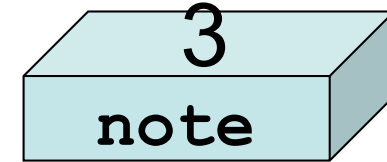
```
somme = somme + note;
```

```
cout << "Entrez la note numero 4" << endl;
```

```
cin >> note;
```

```
somme = somme + note;
```

```
cout << "Moyenne = " << somme / 4 << endl;
```



Pour vérifier le programme précédent, supposons que l'utilisateur entre les notes 1, 2, 3 et 4:

```
float note, somme;
```

```
cout << "Entrez la note numero 1" << endl;
```

```
cin >> note;
```

```
somme = note;
```

```
cout << "Entrez la note numero 2" << endl;
```

```
cin >> note;
```

```
somme = somme + note;
```

```
cout << "Entrez la note numero 3" << endl;
```

```
cin >> note;
```

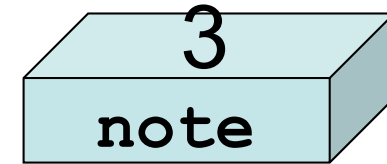
```
→ somme = somme + note;
```

```
cout << "Entrez la note numero 4" << endl;
```

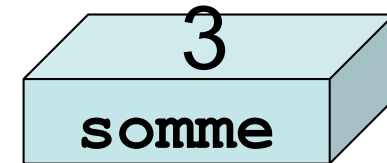
```
cin >> note;
```

```
somme = somme + note;
```

```
cout << "Moyenne = " << somme / 4 << endl;
```



$$3+3=6$$



Pour vérifier le programme précédent, supposons que l'utilisateur entre les notes 1, 2, 3 et 4:

```
float note, somme;
```

```
cout << "Entrez la note numero 1" << endl;
```

```
cin >> note;
```

```
somme = note;
```

```
cout << "Entrez la note numero 2" << endl;
```

```
cin >> note;
```

```
somme = somme + note;
```

```
cout << "Entrez la note numero 3" << endl;
```

```
cin >> note;
```

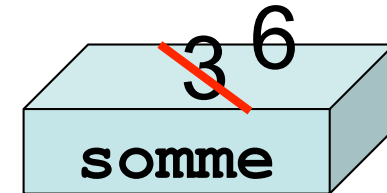
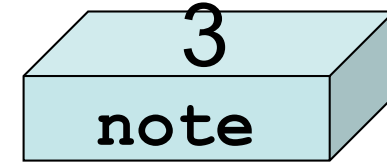
```
→ somme = somme + note;
```

```
cout << "Entrez la note numero 4" << endl;
```

```
cin >> note;
```

```
somme = somme + note;
```

```
cout << "Moyenne = " << somme / 4 << endl;
```



Même programme en utilisant une boucle `for`.
Attention à ne pas oublier d'initialiser `somme` !

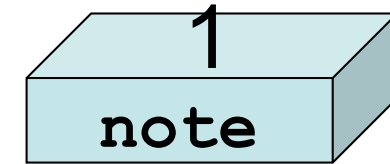
```
float note, somme;

somme = 0;
for(int i = 1; i <= 4; i++)
{
    cout << "Entrez la note numero " << i << endl;
    cin >> note;
    somme = somme + note;
}

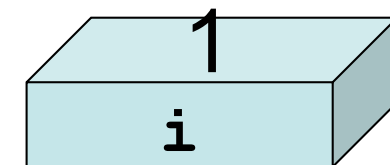
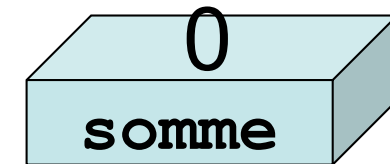
cout << "Moyenne = " << somme / 4 << endl;
```

Même programme en utilisant une boucle `for`.
Attention à ne pas oublier d'initialiser `somme` !

```
float note, somme;  
  
somme = 0;  
for(int i = 1; i <= 4; i++)  
{  
    cout << "Entrez la note numero " << i << endl;  
    cin >> note;  
    somme = somme + note;  
}  
  
cout << "Moyenne = " << somme / 4 << endl;
```



$$0+1=1$$



Même programme en utilisant une boucle `for`.
Attention à ne pas oublier d'initialiser `somme` !

```
float note, somme;
```

```
somme = 0;
```

```
for(int i = 1; i <= 4; i++)
```

```
{
```

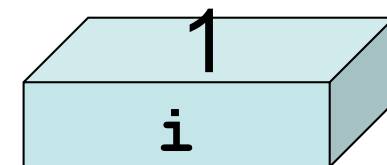
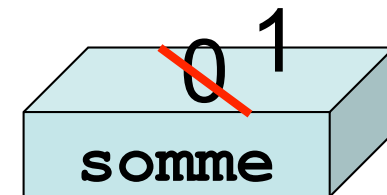
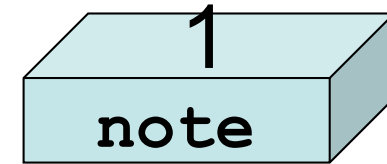
```
    cout << "Entrez la note numero " << i << endl;
```

```
    cin >> note;
```

```
→ somme = somme + note;
```

```
}
```

```
cout << "Moyenne = " << somme / 4 << endl;
```



Comment laisser l'utilisateur choisir le nombre de notes ?

```
float note, somme;
int nombre_de_notes;

cout << "Entrez le nombre de notes" << endl;
cin >> nombre_de_notes;

somme = 0;
for(int i = 1; i <= nombre_de_notes; i++)
{
    cout << "Entrez la note numero " << i << endl;
    cin >> note;
    somme = somme + note;
}

cout << "Moyenne = " << somme / nombre_de_notes << endl;
```

Boucles imbriquées

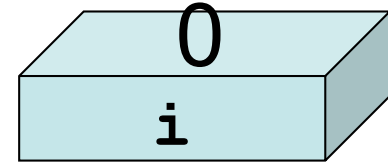
Exemple:

```
for(int i = 0; i < 3; i++)  
{  
    for(int j = 0; j < 2; j++)  
        cout << "i = " << i << ", j = " << j << endl;  
}
```

Boucles imbriquées

Exemple:

```
→ for(int i = 0; i < 3; i++)  
{  
    for(int j = 0; j < 2; j++)  
        cout << "i = " << i << ", j = " << j << endl;  
}
```



Boucles imbriquées

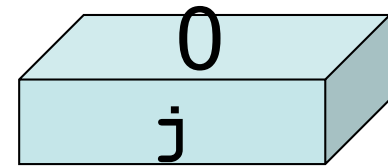
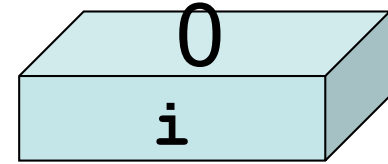
Exemple:

```
for(int i = 0; i < 3; i++)
```

```
{
```

```
→ for(int j = 0; j < 2; j++)  
    cout << "i = " << i << ", j = " << j << endl;
```

```
}
```



Boucles imbriquées

Exemple:

```
for(int i = 0; i < 3; i++)
```

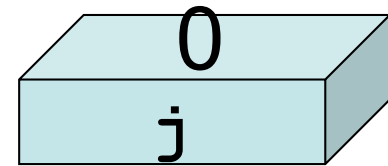
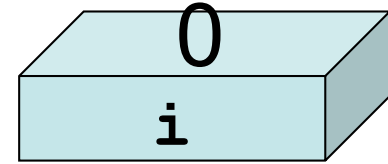
```
{
```

```
    for(int j = 0; j < 2; j++)
```

```
    cout << "i = " << i << ", j = " << j << endl;
```

```
}
```

i = 0, j = 0



Boucles imbriquées

Exemple:

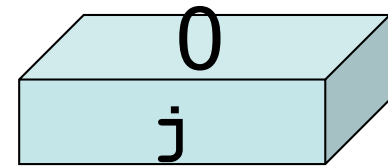
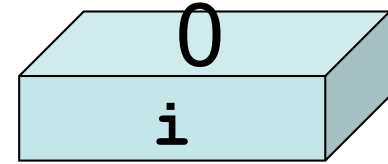
```
for(int i = 0; i < 3; i++)
```

```
{
```

```
→ for(int j = 0; j < 2; j++)  
    cout << "i = " << i << ", j = " << j << endl;
```

```
}
```

```
i = 0, j = 0
```



Boucles imbriquées

Exemple:

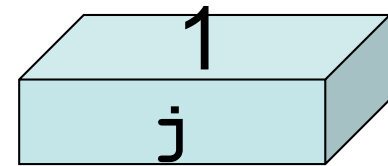
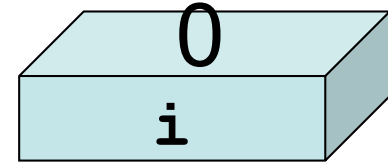
```
for(int i = 0; i < 3; i++)
```

```
{
```

```
→ for(int j = 0; j < 2; j++)  
    cout << "i = " << i << ", j = " << j << endl;
```

```
}
```

```
i = 0, j = 0
```



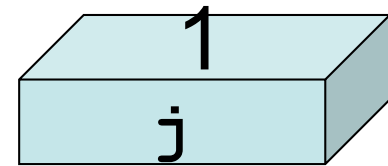
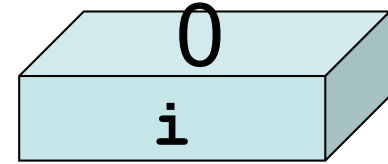
Boucles imbriquées

Exemple:

```
for(int i = 0; i < 3; i++)  
{  
    for(int j = 0; j < 2; j++)  
→ cout << "i = " << i << ", j = " << j << endl;  
}
```

i = 0, j = 0

i = 0, j = 1



Boucles imbriquées

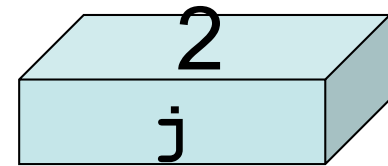
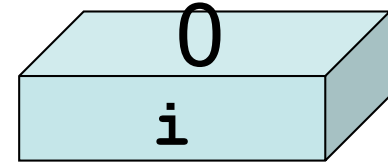
Exemple:

```
for(int i = 0; i < 3; i++)  
{
```

```
→ for(int j = 0; j < 2; j++)  
    cout << "i = " << i << ", j = " << j << endl;  
}
```

```
i = 0, j = 0
```

```
i = 0, j = 1
```



Boucles imbriquées

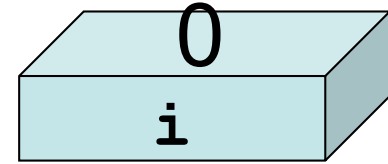
Exemple:

```
for(int i = 0; i < 3; i++)  
{  
    for(int j = 0; j < 2; j++)  
        cout << "i = " << i << ", j = " << j << endl;  
}
```



i = 0, j = 0

i = 0, j = 1



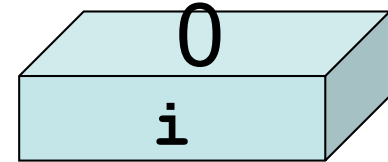
Boucles imbriquées

Exemple:

```
→ for(int i = 0; i < 3; i++)  
  {  
    for(int j = 0; j < 2; j++)  
      cout << "i = " << i << ", j = " << j << endl;  
  }
```

i = 0, j = 0

i = 0, j = 1



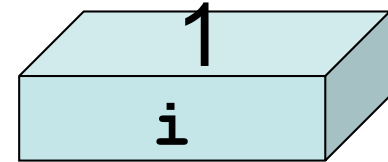
Boucles imbriquées

Exemple:

```
→ for(int i = 0; i < 3; i++)  
  {  
    for(int j = 0; j < 2; j++)  
      cout << "i = " << i << ", j = " << j << endl;  
  }
```

i = 0, j = 0

i = 0, j = 1



Boucles imbriquées

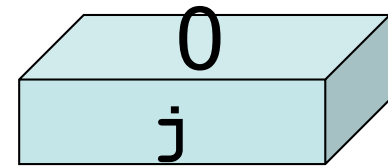
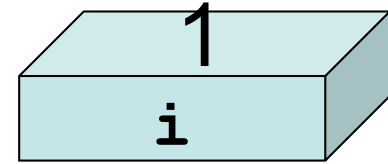
Exemple:

```
for(int i = 0; i < 3; i++)  
{
```

```
→ for(int j = 0; j < 2; j++)  
    cout << "i = " << i << ", j = " << j << endl;  
}
```

i = 0, j = 0

i = 0, j = 1

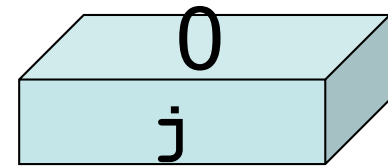
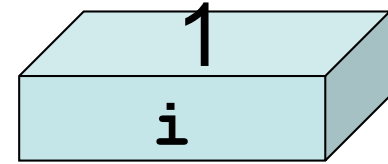


Boucles imbriquées

Exemple:

```
for(int i = 0; i < 3; i++)  
{  
    for(int j = 0; j < 2; j++)  
→ cout << "i = " << i << ", j = " << j << endl;  
}
```

```
i = 0, j = 0  
i = 0, j = 1  
i = 1, j = 0
```



Boucles imbriquées

Exemple:

```
for(int i = 0; i < 3; i++)  
{  
    for(int j = 0; j < 2; j++)  
        cout << "i = " << i << ", j = " << j << endl;  
}
```



```
i = 0, j = 0  
i = 0, j = 1  
i = 1, j = 0  
i = 1, j = 1  
i = 2, j = 0  
i = 2, j = 1
```

Boucles imbriquées

Remarque: Les accolades dans l'exemple précédent ne sont pas nécessaires.

```
for(int i = 0; i < 3; i++)  
{  
    for(int j = 0; j < 2; j++)  
        cout << "i = " << i << ", j = " << j << endl;  
}
```

Compte pour une seule instruction.

peut s'écrire:

```
for(int i = 0; i < 3; i++)  
    for(int j = 0; j < 2; j++)  
        cout << "i = " << i << ", j = " << j << endl;
```

Qu'affichent les programmes suivants ?

D:

```
for(int i = 0; i < 5; i++)
{
    for(int j = 0; j < 5; j++)
        cout << "*";
    cout << endl;
}
```

E:

```
for(int i = 0; i < 5; i++)
{
    for(int j = 0; j < i; j++)
        cout << "*";
    cout << endl;
}
```

F:

```
for(int i = 0; i < 5; i++)
{
    for(int j = 0; j < 5; j++)
        if (i == j)
            cout << "*";
        else
            cout << " ";
    cout << endl;
}
```

Règles d'écriture des programmes

Séparateurs

Les séparateurs sont: l'espace et le retour à la ligne.

Deux identificateurs successifs doivent être séparés par au moins un espace ou un retour à la ligne.

Par exemple, on ne peut pas écrire:

```
intx, y;
```

On peut en revanche écrire:

```
int x, y, z;
```

ou

```
int x, y,
```

```
z;
```

Le format libre

Le langage C autorise une mise en page parfaitement libre. Une instruction peut s'étendre sur un nombre quelconques de lignes, une même ligne peut comporter autant d'instructions que l'on souhaite, etc...

Attention, cette liberté peut vite conduire à des programmes peu lisibles. Voici un exemple de *programme mal présenté*, même s'il est accepté par le compilateur:

```
#include <iostream>
#include <cmath>
using namespace std;

int main(int argc, char * * argv) { float
x; float racine_x;
cout << "Donnez un nombre: " << endl
; cin >> x; if
(x < 0.0)cout<<
"Je ne sais pas calculer la racine de " <<x
;
else {racine_x=
sqrt(x); cout
<< "La racine carree est " << racine_x << endl;}
cout << "Au revoir" << endl; return 0;}
```

Les commentaires

Le compilateur C autorise la présence de commentaires dans les programmes. Il s'agit de textes explicatifs destinés aux lecteurs du programme.

Les commentaires ne changent pas la compilation ni l'exécution du programme.

Commentaires sur plusieurs lignes: Placés entre /* et */.

Commentaires sur une seule ligne: Débutent par //, se terminent à la fin de la ligne.

Exemple page suivante.

Exemple de programme commenté

```
#include <iostream>
using namespace std;
#include <math.h>

/*
  Programme calculant la racine carree d'un nombre
  entre par l'utilisateur.
*/

int main(int argc, char ** argv)
{
  float x;
  float racine_x;

  cout << "Donnez un nombre: " << endl;
  cin >> x;

  if (x < 0.0) // Si le nombre est negatif, on ne peut pas calculer sa racine
    cout << "Je ne sais pas calculer la racine de " << x << endl;
  else
  {
    racine_x = sqrt(x);
    cout << "La racine carree est " << racine_x << endl;
  }

  cout << "Au revoir." << endl;

  return 0;
}
```

Règles d'écriture

1. Une accolade ouvrante { et l'accolade fermante associée } doivent se trouver sur la même colonne.
2. Les instructions appartenant à un même bloc doivent commencer sur la même colonne.
3. On utilisera les commentaires pour clarifier le fonctionnement du programme.

```
int main(int argc, char ** argv)
{
    float x;
    float racine_x;

    cout << "Donnez un nombre: " << endl;
    cin >> x;

    if (x < 0.0)
    | cout << "Je ne sais pas calculer la racine de " << x << endl;
    else
    {
    | racine_x = sqrt(x);
    | cout << "La racine carree est " << racine_x << endl;
    }
    cout << "Au revoir" << endl;
    return 0;
}
```

Règles d'écriture

Emacs indente automatiquement la ligne où se trouve le curseur quand on presse la touche *Tab*.

→ Si la ligne ne s'indente pas correctement, il y a sans doute une erreur de syntaxe aux environs du curseur, en particulier une accolade ou une parenthèse manquante.