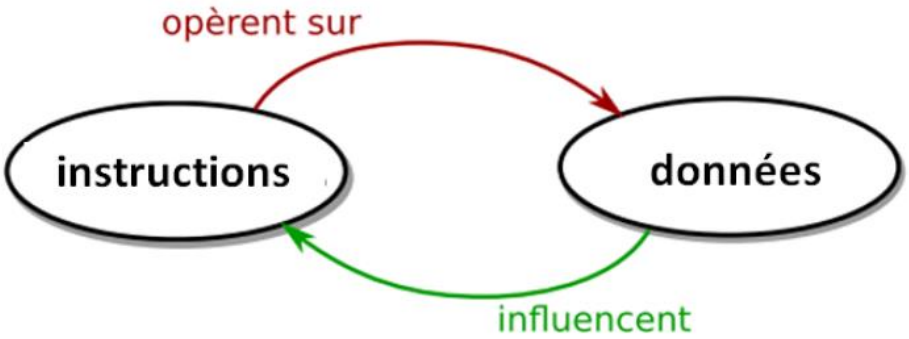
LANGAGE C++

STRUCTURES DE CONTRÔLE

Notion de structure de contrôle

Jusqu'ici, dans un programme,

- Les instructions sont exécutées séquentiellement, c'est-à-dire dans l'ordre où elles apparaissent.
- Les instructions **opèrent sur les données** (les valeurs des variables) grâce à des opérateurs



Les structures de contrôle permettent aux données, à leur tour, **d'influencer les traitements**

Il y a 3 structures de contrôle:

- les branchements conditionnels
- les itérations, et
- les boucles conditionnelles.

Branchement conditionnel

Démo : **Exécution pas à pas, branchement conditionnel ou structure alternative**

1 Structure alternative réduite

```
int main()
{
    int n;
    cout << "Entrez votre nombre:" << endl;
    cin >> n;

    if (n < 5) {
        cout << "condition vraie." << endl;
        cout << "Plus petit que 5." << endl;
    }
    cout << "Au revoir" << endl;
    return 0;
}
```

2 Structure alternative complète

```
int main()
{
    int n;
    cout << "Entrez votre nombre:" << endl;
    cin >> n;

    if (n < 5) {
      cout << "condition vraie." << endl;
      cout << "Plus petit que 5." << endl;
    } else {
      cout << "condition fausse." << endl;
      cout << "Plus grand ou egal a 5." << endl;
    }
    cout << "Au revoir" << endl;
    return 0;
}
```

Note : La notion de bloc – séquence d'instruction représentant une entité

Lorsque l'on veut regrouper plusieurs instructions, nous devons créer ce que l'on appelle un bloc, c'est-à-dire un ensemble d'instructions compris entre les accolades { et }.

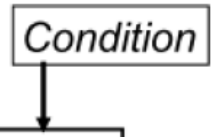
Structure de Choix imbriqués

Exemples :

- x = 1, y = 1, z = 1
- x = 1, y = 1, z = 2
- x = 1, y = 2, z = 1
- x = 1, y = 2, z = 2
- x = 1, y = 2, z = 3

```
if (x == y) {
    if (y == z) {
        cout << "Les trois valeurs sont egales." << endl;
    } else {
        cout << "Seules les deux premieres valeurs sont egales." << endl;
    }
} else {
    if (x == z) {
        cout << "Seules la premiere et la troisieme valeurs sont egales." << endl;
    } else {
        if (y == z) {
            cout << "Seules les deux dernieres valeurs sont egales." << endl;
        } else {
            cout << "Les trois valeurs sont differentes." << endl;
        }
    }
}
```

Les conditions simples




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

Une condition est **toujours** entourée de parenthèses.

- Une condition est un cas particulier d'expression, qui ne peut prendre que deux valeurs.
- En C++, ces valeurs se notent **true** et **false**.
- Une condition simple compare **deux expressions** grâce un opérateur de comparaison

Opérateurs de comparaison

Opérateur de comparaison	Signification
>	inférieur à
<	supérieur à
==	égal à
<=	inférieur ou égal à
>=	supérieur ou égal à
!=	différent de

Exercice : Qu'affichent les séquences d'instructions suivantes ?

```
int a(1);
int b(2);

if (a == b) {
  cout << "Cas 1" << endl;
} else {
  cout << "Cas 2" << endl;
}

if (2 * a == b){
  cout << "b est egal au double de a." << endl;
}
```

```
int a(1);
int b(2);

if (a != b) {
  cout << "Cas 2" << endl;
} else {
  cout << "Cas 1" << endl;
}

if (2 * a != b) {
  cout << "b est different du double de a." << endl;
}
```

```
int a(1);
int b(2);

if (a <= b) {
  cout << "Cas 3" << endl;
} else {
  cout << "Cas 4" << endl;
}

if (2 * a <= b) {
  cout << "b est superieur ou egal au double de a." << endl;
}
```

Les opérateurs logiques

Les opérateurs logiques permettent de relier des conditions simples et ainsi créer des conditions composées (complexes).

L'opérateur logique ET : (a < b) && (c < d)

=> est vraie **uniquement** si les deux conditions (a < b) et (c < d) sont toutes les deux vraies.

Exercice : donner les résultats de l'exécution de la séquence suivante pour n ayant comme valeur 0, 4, 8, 10 et 12.

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

L'opérateur logique OU : (a < b) || (c < d)

=> est vraie si **au moins** une des deux conditions (a < b) ou (c < d) est vraie.

Exercice : donner le résultat de l'exécution de la séquence suivante pour n et m ayant comme valeur : (-1,-1), (-1,1), (0,0) et (1,1)

```
if ((m >= 0) or (n >= 0)) {
    cout << "au moins une valeur est positive" << endl;
} else  {
    cout << "les deux valeurs sont negatives" << endl;
}
```

L'opérateur logique NON: **! (a < b)**

=> est vraie si (a < b) est fausse, et fausse si (a < b) est vraie.

Note : opérateur Non est un opérateur unaire, il n'attend qu'un seul opérande.

Quelques erreurs classiques

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

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

2.

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

- a vaut 1 est toujours affiché quelle que soit la valeur de a!
- Le point-virgule est considéré comme une instruction, qui ne fait rien.

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

```
if (n < p)
  cout << "n est plus petit que p" << endl;
  max = p;
else
  cout << "n est plus grand ou egal a p" << endl;
return 0;
```

```
Message
=== Build: Debug in test6 (compiler: GNU GC
In function 'int main()':
error: 'else' without a previous 'if'
=== Build failed: 1 error(s), 0 warning(s)
```

Quiz

```cpp
#include <iostream>
using namespace std;
int main()
{
    int n = 0, p = 0;
    cout << "Entrez le premier nombre:" << endl;
    cin >> n;
    cout << "Entrez le deuxieme nombre:" << endl;
    cin >> p;

    if ((n < p) and (2 * n >= p)) {
      cout << "1";
    }

    if ((n < p) or (2 * n >= p)) {
      cout << "2";
    }

    if (n < p) {
      if (2 * n >= p) {
        cout << "3";
      } else {
        cout << "4";
      }
    }
    cout << endl;

    return 0;
}
```

A: 2
B: 24
C: 123
D: 1234

Qu'affiche ce programme quand l'utilisateur saisit 1 et 2, 1 et 3, 2 et 1 ?

Itération – concept

Imaginons que l'on veuille affiche la table de multiplication de 5.

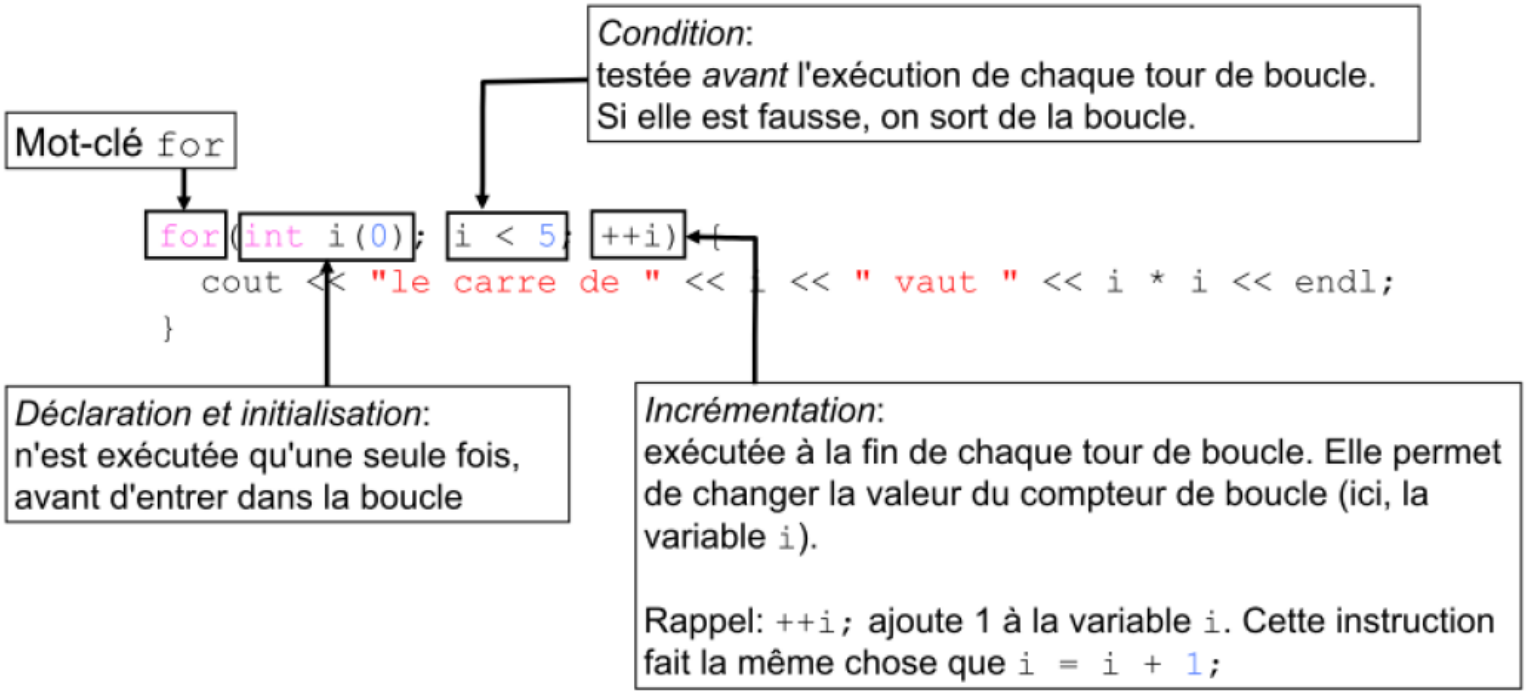
```
cout << "5 multiplie par 1  vaut " << 5 * 1  << endl;
cout << "5 multiplie par 2  vaut " << 5 * 2  << endl;
cout << "5 multiplie par 3  vaut " << 5 * 3  << endl;
cout << "5 multiplie par 4  vaut " << 5 * 4  << endl;
cout << "5 multiplie par 5  vaut " << 5 * 5  << endl;
cout << "5 multiplie par 6  vaut " << 5 * 6  << endl;
cout << "5 multiplie par 7  vaut " << 5 * 7  << endl;
cout << "5 multiplie par 8  vaut " << 5 * 8  << endl;
cout << "5 multiplie par 9  vaut " << 5 * 9  << endl;
cout << "5 multiplie par 10 vaut " << 5 * 10 << endl;
```

est équivalent à

Corps de la boucle:
Bloc d'instructions (seront exécutées à chaque tour de boucle).

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

Itération – la boucle for



Une boucle for permet de **répéter** un nombre donné de fois la même séquence d'instructions (**bloc**).

- On utilise une **variable de contrôle d'itération** caractérisée par sa **valeur initiale**, sa valeur **finale**, son **pas de variation**.
- Elles sont utilisées quand le nombre de répétitions est **connu** avant d'entrer dans la boucle

Note : Dans cette forme, La durée de vie de la variable d'itération i est la boucle.

Syntaxe de l'instruction for

```
for(déclaration_et_initialisation; condition; incrémentation) {
  bloc
}
```

L'initialisation, la condition, et l'incrémentation sont séparées par des points-virgules

```
for(int i(0); i < 5; ++i) {
  cout << "le carre de " << i << " vaut " << i * i << endl;
}
```

Corps de la boucle:
Bloc d'instructions qui seront exécutées à chaque tour de boucle.

Exercice : Exécution pas à pas - Que s'affiche-t-il quand on exécute le programme ?

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

A: 0p 1 2p 3 4p
B: 0p 1 2 3 4
C: 0 1 2p 3 4
D : 0p 1p 2p 3p 4p

Exemples d'autres formes de boucles for

```
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),j(5); i < 5, j>0; ++i,--j){
    cout << "(" << i << ", " << j << ")   ";
}
```

```
( 0, 5)  ( 1, 4)  ( 2, 3)  ( 3, 2)  ( 4, 1)
Process returned 0 (0x0)   execution time : 0.005 s
Press any key to continue.
```

Boucles infinies et erreur classiques

Boucles infinies : La boucle for peut ne pas s'arrêter, ce qui se produit quand la condition est toujours vraie.

1. On s'est trompé sur la condition

```
for(int i(0); i > -1; ++i) { // !!!
    ...
}
```

2. On s'est trompé sur l'incrémentation

```
for(int i(0); i < 10; ++j) { // !!!
    ...
}
```

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

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

```
bonjour

Process returned 0 (0x0)   execution time : 0.010 s
Press any key to continue.
```

4. Attention aux accolades

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

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

Process returned 0 (0x0)   execution time : 0.030 s
Press any key to continue.
```

Exemple : Moyenne de 4 notes

```
double note, somme(0.0);

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

Répétition de la même séquence d'instructions.

Exercice : Comment modifier le code pour laisser l'utilisateur choisir le nombre de notes ?

Programme équivalent avec la boucle for

```
double note, somme(0.0);

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

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

Boucles imbriquées

Reprenons l'exemple précédent de la table de multiplication par 5 :

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

Affiche 9 fois la table de multiplication par 5

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

Affiche la table de multiplication par 1, puis par 2, jusque 10.

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

Programme complet

```cpp
#include <iostream>
using namespace std;
int main()
{
    for(int j(2); j <= 10; ++j) {
      cout << "Table de multiplication par " << j << ":" << endl;
      for(int i(1); i <= 10; ++i) {
        cout << j << " multiplie par " << i << " vaut " << j * i << endl;
      }
    }
    return 0;
}
```

```
Table de multiplication par 2:
2 multiplie par 1 vaut 2
2 multiplie par 2 vaut 4
2 multiplie par 3 vaut 6
2 multiplie par 4 vaut 8
2 multiplie par 5 vaut 10
2 multiplie par 6 vaut 12
2 multiplie par 7 vaut 14
2 multiplie par 8 vaut 16
2 multiplie par 9 vaut 18
2 multiplie par 10 vaut 20
Table de multiplication par 3:
3 multiplie par 1 vaut 3
3 multiplie par 2 vaut 6
3 multiplie par 3 vaut 9
3 multiplie par 4 vaut 12
3 multiplie par 5 vaut 15
3 multiplie par 6 vaut 18
3 multiplie par 7 vaut 21
3 multiplie par 8 vaut 24
3 multiplie par 9 vaut 27
3 multiplie par 10 vaut 30
Table de multiplication par 4:
4 multiplie par 1 vaut 4
4 multiplie par 2 vaut 8
4 multiplie par 3 vaut 12
4 multiplie par 4 vaut 16
4 multiplie par 5 vaut 20
4 multiplie par 6 vaut 24
4 multiplie par 7 vaut 28
4 multiplie par 8 vaut 32
4 multiplie par 9 vaut 36
4 multiplie par 10 vaut 40
Table de multiplication par 5:
5 multiplie par 1 vaut 5
5 multiplie par 2 vaut 10
5 multiplie par 3 vaut 15
5 multiplie par 4 vaut 20
5 multiplie par 5 vaut 25
5 multiplie par 6 vaut 30
5 multiplie par 7 vaut 35
5 multiplie par 8 vaut 40
5 multiplie par 9 vaut 45
5 multiplie par 10 vaut 50
Table de multiplication par 6:
6 multiplie par 1 vaut 6
6 multiplie par 2 vaut 12
6 multiplie par 3 vaut 18
6 multiplie par 4 vaut 24
6 multiplie par 5 vaut 30
6 multiplie par 6 vaut 36
6 multiplie par 7 vaut 42
6 multiplie par 8 vaut 48
6 multiplie par 9 vaut 54
6 multiplie par 10 vaut 60
Table de multiplication par 7:
7 multiplie par 1 vaut 7
7 multiplie par 2 vaut 14
7 multiplie par 3 vaut 21
7 multiplie par 4 vaut 28
7 multiplie par 5 vaut 35
7 multiplie par 6 vaut 42
7 multiplie par 7 vaut 49
7 multiplie par 8 vaut 56
7 multiplie par 9 vaut 63
7 multiplie par 10 vaut 70
Table de multiplication par 8:
8 multiplie par 1 vaut 8
8 multiplie par 2 vaut 16
8 multiplie par 3 vaut 24
8 multiplie par 4 vaut 32
8 multiplie par 5 vaut 40
8 multiplie par 6 vaut 48
8 multiplie par 7 vaut 56
8 multiplie par 8 vaut 64
8 multiplie par 9 vaut 72
8 multiplie par 10 vaut 80
Table de multiplication par 9:
9 multiplie par 1 vaut 9
9 multiplie par 2 vaut 18
9 multiplie par 3 vaut 27
9 multiplie par 4 vaut 36
9 multiplie par 5 vaut 45
9 multiplie par 6 vaut 54
9 multiplie par 7 vaut 63
9 multiplie par 8 vaut 72
9 multiplie par 9 vaut 81
9 multiplie par 10 vaut 90
Table de multiplication par 10:
10 multiplie par 1 vaut 10
10 multiplie par 2 vaut 20
10 multiplie par 3 vaut 30
10 multiplie par 4 vaut 40
10 multiplie par 5 vaut 50
10 multiplie par 6 vaut 60
10 multiplie par 7 vaut 70
10 multiplie par 8 vaut 80
10 multiplie par 9 vaut 90
10 multiplie par 10 vaut 100

Process returned 0 (0x0)   execu
Press any key to continue.
```

Quizz

```
for(int i(0); i < 3; ++i) {
  for(int j(0); j < 4; ++j) {
    if (i == j) {
      cout << "*";
    } else {
      cout << j;
    }
  }
  cout << endl;
}
```

A:
*123
*123
*123

B:
012*
012*
012*

C:
****
****
****

D:
*123
0*23
01*3

?

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

A:

0
01

B:
0
01
012

C:
rien

D:
0123
0123
0123

?

Les boucles conditionnelles - concept

Elles sont utilisées quand le nombre de répétitions est **inconnu** avant d'entrer dans la boucle,

- Les boucles conditionnelles permettent de répéter une séquence d'instruction définit par un **bloc**
- L'itération se fera par une **condition**.

Tant que la condition est vraie nous itérons.

Comment forcer l'utilisateur à entrer une note supérieure à 0 ?

```
int nombre_de_notes;

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

Exécuter le bloc d'instruction

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

Tant que la valeur saisie n'est pas valide, on itère

Les structures do…while et while

Deux structures de boucle conditionnelle

Syntaxes

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

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

Le bloc d'instructions de la structure **do…while** sont toujours exécutées **au moins une fois**.

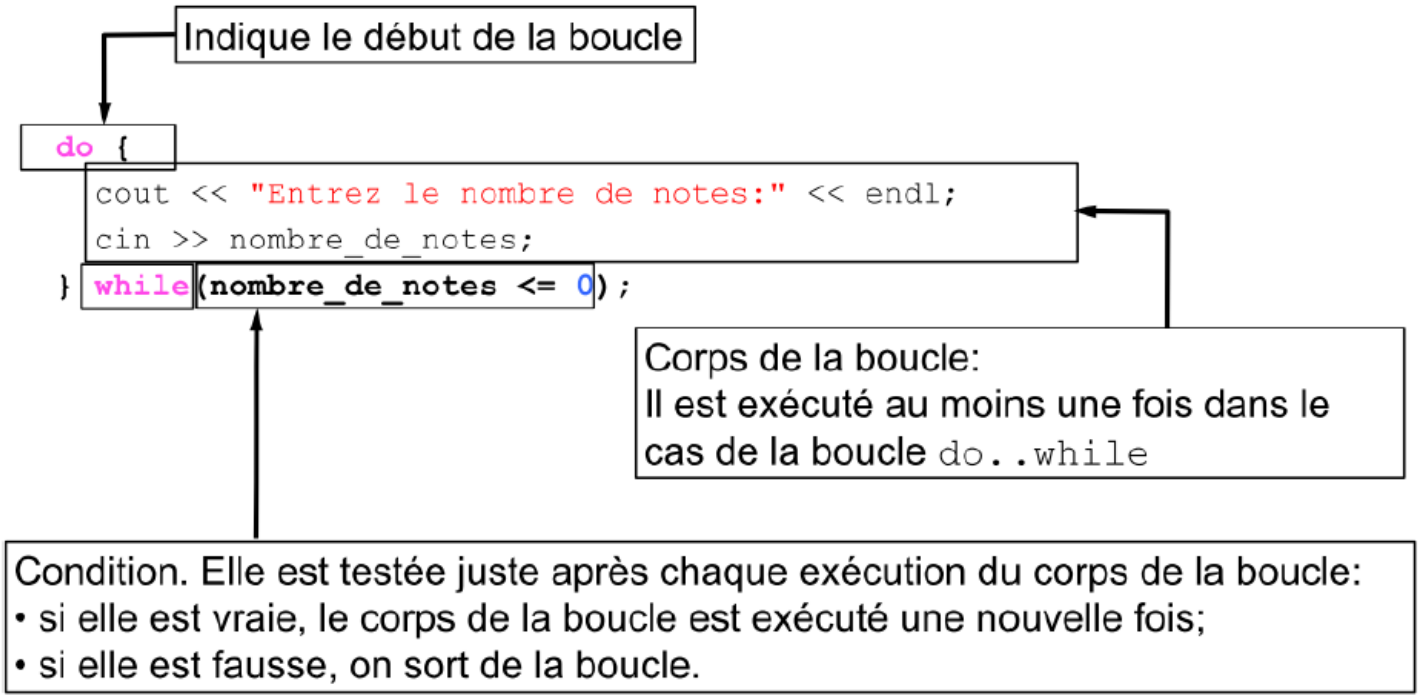
- Pour la structure **while,** la condition est testée **avant** d'entrer dans la boucle (le bloc peut ne jamais être exécuté).
- La condition peut utiliser des opérateurs logiques.
- Les parenthèses autour de la condition sont obligatoires.
- Si la condition ne devient jamais fausse, les instructions dans la boucle sont répétées indéfiniment !

Démo : Forcer l'utilisateur à saisir un nombre de note supérieur à 0

```
int nombre_de_notes = 0;
do {
     cout << "Entrez le nombre de notes:" << endl;
     cin >> nombre_de_notes;
} while(nombre_de_notes <= 0);
```

Répéter les instructions suivantes tant que la réponse n'est pas valide. Le nombre d'itération n'est pas connu

L'instruction do…while



Exercice : Que s'affiche-t-il quand on exécute les séquences d'instructions suivantes?

```
int i(100);
do {
  cout << "bonjour" << endl;
} while (i < 10);
```

```
int i(100);
while (i < 10) {
  cout << "bonjour" << endl;
}
```

Quelques erreurs classiques

Il n'y a pas de ; à la fin de la condition du **while**

```
int i = 0;
while (i < 10); // Boucle infinie
   ++i;
```

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

```
int i = 0;
do {
  ++i;
} while(i < 10);
```

Choix boucle for / while / do…while

Utiliser la boucle **for** quand le nombre d'itérations (de répétitions) est connu avant d'entrer dans la boucle :

```
for(int i(0); i < nombre_d_iterations; ++i) {
    instructions;
}
```

Sinon, utiliser une structure boucle conditionnelle

- Utiliser la boucle **do…while** quand les instructions doivent être effectuées au moins une fois

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

- Sinon, utiliser la forme **while**

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

Démo : **Améliorer le programme « lire le nombre de note »**

- avertir l'utilisateur de saisie erroné ("il faut entrer un nombre supérieur à 0")
- ne pas dupliquer le test (nombre_de_notes <= 0)

Programme complet

```cpp
#include <iostream>
using namespace std;
int main()
{
    int nombre_de_notes = 0;
    bool saisieInvalide = false;
    do {
      cout << "Entrez le nombre de notes:" << endl;
      cin >> nombre_de_notes;
      saisieInvalide = (nombre_de_notes <= 0);
      if (saisieInvalide) {
        cout << "il faut entrer un nombre supérieur a 0" << endl;
      }
    } while(saisieInvalide);
    return 0;
}
```

```
Entrez le nombre de notes:
0
il faut entrer un nombre supÚrieur a 0
Entrez le nombre de notes:
-1
il faut entrer un nombre supÚrieur a 0
Entrez le nombre de notes:
2

Process returned 0 (0x0)   execution ti
Press any key to continue.
```

Comment trouver la condition ?

Exemple :

- Comment forcer l'utilisateur à entrer une note supérieure à 0 et inférieur ou égal à 10 ?
- On veut répéter la boucle **tant que le nombre de notes est incorrect**
- Le nombre de notes est incorrect si il est **inférieur ou égal à 0 ou si il est supérieur à 10**,

```
(nombreDeNotes <= 0 || nombreDeNotes > 10)
```

Exemple : Programme qui demande à l'utilisateur 1 de deviner un entier compris entre 1 et 10 et qui aura été saisi par l'utilisateur 2.

Etude de cas : **Le nombre mystère**

Les blocs

En C++, les instructions peuvent être regroupées en blocs.

- Les blocs sont identifiés par des délimiteurs de début et de fin : { et }
- Ils peuvent contenir leurs propres déclarations et initialisation de variables:

```
if (i != 0) {
  int j(0);
  ...
  j = 2 * i;
  ...
}
// A partir d'ici, on ne peut plus utiliser j
```

Notion de portée

Les variables déclarées à l'intérieur d'un bloc sont appelées **variables locales** (au bloc).

Elles ne sont accessibles qu'à l'intérieur du bloc.

- Les variables déclarées en dehors de tout bloc (même du bloc main(){}) sont appelées variables globales (au programme). Elles sont accessibles dans l'ensemble du programme. **A EVITER**
- La **portée** d'une variable, c'est l'ensemble des lignes de code où cette variable est accessible, autrement dit où elle est définie, existe, a un sens donc un espace mémoire réservé.

Bonne pratique :

Déclarer les variables au plus près de leur utilisation

```
if (i != 0) {
    int j(0);

    ...
    j = 2 * i;
    ...
}
```

plutôt que

```
int j(0);
if (i != 0) {

    ...
    j = 2 * i;
    ...
}
```

Notion de portée

Règles de résolution de portée

En cas d'ambiguïté, c'est-à-dire quand plusieurs variables de portées différentes portent le même nom: la variable « la plus proche » est choisie (en terme de bloc)

Portée : cas des itérations

La déclaration d'une variable à l'intérieur d'une itération est une déclaration locale au bloc de la boucle, et aux deux instructions de test et d'incrément.

```
int main()
{
    for(int i(0); i < 5; ++i) {
        cout << i << endl;
    }
    // A partir d'ici, on ne peut plus utiliser ce i
    i = i + 1;
    return 0;
}
```