Chapitre 4 – Le flux de l'exécution
Section 2 Les instructions conditionnelles

Les instructions conditionnelles

1. Conditions

Vrai et faux en c/c++

Valeurs : 0 pour faux.
Les types booléens

Les opérateurs relationnels

== (égalité)
!= (inégalité)
< (plus petit que)
> (plus grand que)

Combinaisons

&& (ET)
|| (OU)
! (NOT)
précédence

2. Structures de décision

Structures

if
if/else
if/else if/else
switch/case

3. Structures de répétition (boucles)

Structures de répétition

for
while
do/while

Techniques d'arrêt

condition
break
goto

Analyse détaillée

détermination des bornes
tableaux de valeurs

Les conditions d'exécution

Nous avons prétendu jusqu'à maintenant que l'exécution d'un programme était séquentielle, une instruction après l'autre, dans le domaine de validité du main(), soit entre les accolades { }. Voici la première exception à ce principe : des instructions qui ne seront exécutées qu'à certaine(s) condition(s). Si la condition globale est VRAI, le bloc d'instruction dépendant de cette condition peut être exécuté, sinon il n'est pas exécuté.

Vrai ou faux ?

Il importe donc de savoir évaluer la véracité d'une expression conditionnelle, pour pouvoir en créer une appropriée. Les entités donnant des valeurs vraies ou fausses sont les opérateurs de comparaisons, les variables elles-mêmes et les valeurs de retour de fonctions.

Les variables sont VRAIES si elles ne sont pas zéro, à zéro elles sont fausses. Même les valeurs négatives, les caractères non null et les adresses assignées à un pointeur sont considérés comme VRAI dans une condition.

Dans le langage C, il n'existe pas de type bool. Cependant, il est possible de créer un type booléen dans notre code par l'utilisation de l'instruction **typedef**. À cet effet, une variable de type char est utilisée parce qu'elle est celle prenant le moins d'espace (un octet ou 8 bits). Parfois certains programmeurs typedef un int à la place. Pourtant, les valeurs booléennes possibles ne sont que 0 et 1, au nombre de deux. Un seul bit serait suffisant mais la plus petite unité de mémoire accessible physiquement est l'octet, donc prenons un char.

```
typedef unsigned char boolean
#define TRUE 1
#define FALSE 0
boolean a = TRUE;
```

ou encore

```
typedef enum {mFALSE=0; mTRUE} mon_bool;
mon_bool a = mTRUE;
```

En C++, le type bool existe depuis peu, il est implémenté à l'aide d'une classe et il est possible de lui assigner des nombres 0 ou 1 ou même d'autres nombres qui seront convertis en 1 automatiquement ou encore le résultat d'une comparaison converti en 0 ou 1 automatiquement Il est également possible, à la place d'utiliser la notation nombre, de lui assigner les mots-clés **true** et **false**. Lors de l'affichage de la valeur de la variable, ces deux alternatives sont également présente selon l'option de formatage choisie.

```
bool a = true;
bool b = 3 > 10;  // reçoit false, c'est à dire 0

if( a )
{
  …
}

if( b )
{
  …
}
```

Les opérateurs de comparaisons

	Opérateurs de comparaison (égalité et relationnels)
==	Égalité
!=	Inégalité
<	Plus petit que
<=	Plus petit ou égal à
>	Plus grand que
>=	Plus grand ou égal à

Les opérateurs de comparaison retournent vrai quand l'état des opérandes vérifie leur signification. Par exemple, l'opérateur de comparaison d'égalité, ==, retourne VRAI lorsque les deux opérandes placée autour de lui sont parfaitement identiques. Un autre exemple est l'opérateur < qui retourne VRAI quand son opérande de gauche est plus petite que son opérande de droite.

Ne jamais comparer de variables réelles (float, double) pour l'égalité (==) ou l'inégalité (! =).

La relation d'égalité est difficilement vraie sachant que les variables réelles ne sont que des approximations de nombres réels, il est fort improbable d'obtenir la valeur attendue. Son utilisation crée des blocs qui ne seront jamais exécutés.
if(v == 10.0f) … // jamais le corps du if ne sera exécuté.

Au contraire, la relation d'inégalité sera toujours vraie. Son utilisation cause souvent des boucles infinies.
while(v ! = 10.0f) … cette boucle tourne pour toujours.

Nous serons avisés de choisir les comparaisons de seuils, c'est à dire plus petit ou plus grand que
while(v < 10.0f) … cette boucle cessera si v est bien incrémenté.

Pour minimiser les erreurs d'oubli d'un = avec l'opérateur = =, une bonne pratique de programmation est de positionner ce qui n'est pas variable à la gauche de l'opérateur.

L'assignement générera une erreur de compilation si l'opérande de gauche n'est pas une variable.

```
if( 0 == variable ) … // Aucune erreur ici, la condition est vérifiée,
                      // voir ce qui se passe si on oublie un = dans cette disposition
if( 0 = variable ) …  // 0 est une lvalue illégale pour l'assignement,
                      // l'opérande de gauche doit être une variable avec l'opérateur =,
                      // donc avec cette disposition, le compilateur détectera tout de suite une erreur.
```

Au contraire, si la disposition permet l'assignement, la fréquente erreur de l'oubli d'un symbole = causera une erreur indétectable par le compilateur, l'erreur affectera la logique du programme.

```
if( variable == 0 ) … // Aucune erreur ici, la condition est vérifiée,
                      // voir ce qui se passe si on oublie un = dans cette disposition
if( variable = 0 ) …  // Aucune erreur de compilation, la variable se voit assigner 0
                      // et le corps du if n'est pas exécuté peu importe la valeur initiale de variable
```

Les combinaisons

À partir de différentes expressions ayant au préalable une valeur vraie ou fausse, il est possible de les combiner selon que l'on désire les deux conditions vraies pour exécuter (ET symbolisé par &&) ou au moins l'une des deux (OU symbolisé par ||). On peut aussi vouloir une condition fausse comme condition d'exécution, dans ce cas, on l'inverse avec le NOT (symbolisé par ! devant la valeur à inverser) pour obtenir vrai lorsque c'est faux initialement.

Table de vérité pour le NOT

A	!A
F	V
V	F

(inversion)

Table de vérité pour le ET

A	B	A && B
F	F	F
F	V	F
V	F	F
V	V	V

Table de vérité pour le OU

A	B	A || B
F	F	F
F	V	V
V	F	V
V	V	V

L'exécution nécessite que l'expression inversée soit initialement fausse.

Par exemple
```
while(!found)
{
  if(tab[j++] = = key)
     found = 1;
}

if( !(a == b))
```
est équivalent à
```
if(a != b)
```

L'exécution nécessite que : les deux valeurs soient VRAI.

Par exemple,
```
while( j<30 && !found )
{
  if(tab[j] = = key)
          found = 1;
  j++;
}

if( a>=20 && a<30 )
{
  cout << "Intervalle 20-30";
}
```

L'exécution nécessite que : au moins une des deux valeurs soit vraie. Il est aussi valide que les deux soient vraies.

Par exemple,
```
if(a<20 || a>=30)
{
  cout << "Exterieur de"
  << " l'intervalle 20-30";
}
```

Table de vérité pour le NOT

A	!A
0	1
1	0

(inversion)

Table de vérité pour le ET

A	B	A && B
0	0	0
0	1	0
1	0	0
1	1	1

Table de vérité pour le OU

A	B	A || B
0	0	0
0	1	1
1	0	1
1	1	1

Il est intéressant de noter que la lecture des deux conditions n'est pas toujours nécessaire pour savoir le résultat et que l'opérande gauche est toujours évaluée en premier. La fausseté du premier élément d'une combinaison ET rend toute la combinaison fausse et la vérification de la seconde expression n'est plus nécessaire. Un programmeur ingénieux utilisera ce fait pour accélérer l'exécution du programme, la première expression devant un opérateur && (ET) sera celle le plus souvent fausse. La véracité du premier élément d'une combinaison OU rend toute la combinaison vraie et la vérification de la seconde expression n'est plus nécessaire pour connaître le résultat. Un programmeur ingénieux utilisera cet autre fait pour optimiser le programme, il placera devant l'opérateur || (OU) ce qui a le plus de chance d'être vrai.

Table de vérité pour le NOT

A	!A
0	non-zéro
non-zéro	0

(inversion)

L'exécution nécessite que l'expression inversée soit initialement fausse.

Par exemple
```
while(!trouve)
{
  if(tab[j++] = = cle)
     trouve = 1;
}
```

```
if( !(a == b) )
est équivalent à
if(a != b )
```

Table de vérité pour le ET

A	B	A && B
0	0	0
0	non-zéro	0
non-zéro	0	0
non-zéro	non-zéro	1

L'exécution nécessite que :
les deux valeurs soient VRAI.

Par exemple,
```
if( a>=20 && a<30 )
{
  cout << "Intervalle 20-30";
}
```

```
while( j<30 && !trouve )
{
  if(tab[j] = = cle)
        trouve = 1;
  j++;
}
```

Table de vérité pour le OU

A	B	A || B
0	0	0
0	non-zéro	1
non-zéro	0	1
non-zéro	non-zéro	1

L'exécution nécessite que :
au moins une des deux valeurs soit VRAI. Il est aussi valide que les deux soient VRAI.

Par exemple,
```
if(a<20 || a>=30)
{
  cout << "Exterieur de"
  << " l'intervalle 20-30";
}
```

La précédence des opérateurs

! > *** / % +** > == != > < <= > >= > && > || > =

La priorité par défaut permet de faire les opérations classiques de comparaisons puis de combinaison des résultats sans utiliser de parenthèses.

Par exemple,

if(a + h < b && c = = d) ? if(((a+h) <b) && (c= =d)).

Il peut arriver que l'usage de parenthèses s'avère nécessaire, si le programmeur veut assigner un résultat à une variable puis comparer cette variable, le tout en une seule ligne.

Par exemple,

while((c = getc()) != '\n') { } ? c = getc(); while(c != '\n') { c = getc(); }

mais while(c = getc() != '\n') { } génerera une erreur ainsi que while('\n' != c = getc()) { }

Pour la précédence de la négation, il faut également faire attention. De priorité très élevée, ! permet d'inverser une seule variable sans problème mais pour inverser le résultat d'un opérateur relationnel il faudrait utiliser les parenthèses.

Par exemple,

if(!(a==b)) ? if(!a = = b)

Cependant, chaque opérateur relationnel a un inverse de sorte qu'une telle combinaison d'opérations est inutile.

if(!(a == b)) ? if(a != b)

if(!(a < b)) ? if(a >= b)

if(!(a > b)) ? if(a <= b)

if(!(a!=b)) ? if(a == b)

if(!(a>=b)) ? if(a< b)

if(!(a<=b)) ? if(a > b)

Opérateur	*Opérateur inverse*
= =	!=
>	<=
<	>=

La précédence de la négation convient parfaitement aux combinaisons d'inverses de variables, pour tester si elles sont à zéro ou non.

while(!trouve && !fini) { }

Comme en logique booléenne, le ET (&&) a précédence sur le OU (||) et en c/c++ les conditions sont évaluées de gauche à droite. Par exemple,

if(a + h < b && c = = d || !g && f > h) ? if(**(**a + h < b && c = = d**)** || **(**!g && f > h**)**)

if(a + h < b || c = = d && !g || f > h) ? if(((a + h < b) || (c = = d && !g)) || (f > h))

Choix de l'instruction de contrôle

Nous verrons dans les pages suivantes diverses structures de contrôle. Ainsi nous apprendrons les structures de sélection tel **if** et les structure itératives (boucles) tel **for**, **while** et **do-while**.

Il incombera à l'étudiant d'apprendre à choisir la bonne structure selon ses besoins. Chaque instruction rend possible un certain nombre d'exécution du bloc d'instructions entre accolades. Par exemple, avec une instruction **if**, il sera possible que le bloc d'instruction soit exécuté une fois ou pas du tout, donc 1 ou 0. Pour une boucle **do-while**, on sait que le bloc d'instructions est certainement exécuté au moins une fois, et qu'il peut y avoir répétition ou non. Pou une boucle **while**, il se peut que le bloc d'instructions ne soit jamais exécuté si la condition est fausse dès le début, c'est la différence avec le do-while, la condition est testé avant toute exécution. Pour une boucle **for**, le nombre d'itération devrait normalement être déterminé à l'avance. Voici un tableau qui résume le nombre d'itération (nombre d'exécution du bloc d'instructions) associé à chaque instruction de contrôle. Il aide ainsi à choisir la bonne instruction pour un problème donné.

if	while	for	do-while
1 ou 0	0 à ?	N	1 à ?

Par exemple, si on sait qu'il nous faut itérer pour toutes les cases d'un damier, et que ce nombre de cases est connu, on doit donc itérer pour les N cases et la structure **for** est de mise. Si on doit plutôt traiter toutes les lignes d'un fichier à mesure qu'on les lit, et que ce nombre de lignes est inconnu, il nous faut travailler TANT QUE il y a des lignes à lire, donc on utilise **while**. Si on doit plutôt demander des informations à un usager, les valider, si elles sont valides arrêter, sinon continuer à les demander tant que ces données ne sont pas valides, c'est une opération qui doit être effectuée au moins une fois, et peut-être plusieurs autres fois, donc c'est un **do-while**. Si on doit jouer à un jeu, et redemander à la fin si l'usager veut encore jouer et boucler le cas échéant, il faut encore utiliser un **do-while**.

La formulation en langage naturel (français, anglais,…) de la solution permet de mieux choisir l'instruction informatique. Voici la 'traduction' de chaque instruction en pseudocode

IF	SI
FOR	POUR
WHILE	TANT QUE
DO-WHILE	FAIRE et CONTINUER TANT QUE

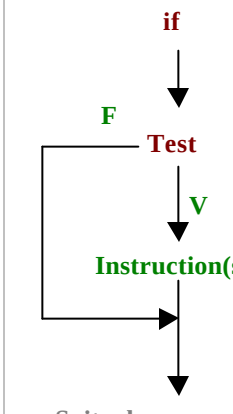
Par exemple, voyons la nuance de formulation pour un bloc qui bouclerais 30 fois:

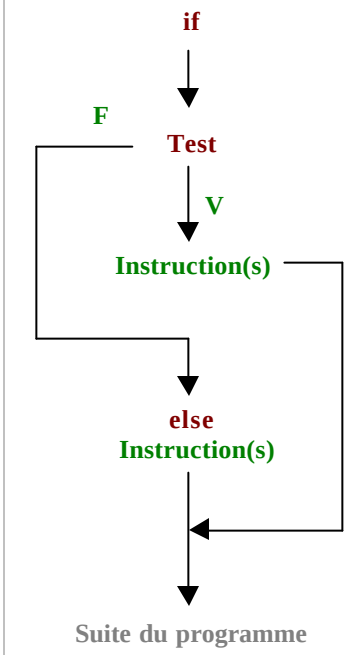
While	Tant que pas 30
For	Pour de 0 à 29

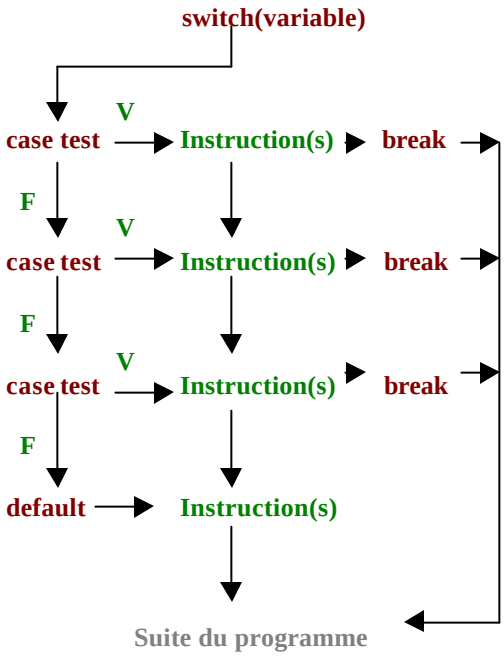
Les structures de décision

Test If	*Test If/else*	*Test Switch/case*
Le bloc sera exécuté 0 ou 1 fois.	Exactement un des deux blocs sera exécuté une seule fois.	Un seul case sera reconnu et l'exécution poursuit jusqu'au break.
if(condition) *{* * instruction;* *}*	*if(condition)* *{* * instruction;* *}* *else* *{* * instruction;* *}*	*switch(variable)* *{* * case valeur : instructions; break;* * case valeur : instructions; break;* * case valeur : instructions; break;* * default : instructions;* *}*

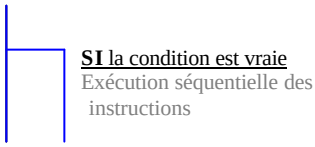
Diagrammes de flux (flow diagrams)

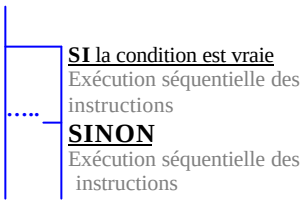


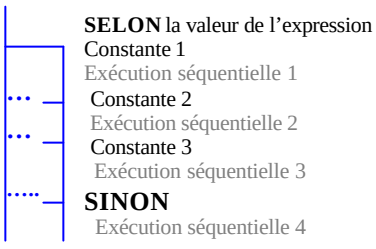




Pseudocode







* Style du pseudocode selon l'enseignement de l'École Polytechnique de Montréal

La structure de sélection if

if (condition **)** **{** bloc d'instructions dépendant de la condition; **}**

Condition testée avant d'exécuter le bloc d'instruction, si la condition est vraie, exécution !

```
// Ce programme affiche si la valeur est 3

int main(void)
{
  int a = 3;
  int b = 5;

 // Sélection
  if( a = = 3 )
  {                    // executé si le == est vrai
    cout << "\n A est 3 ";
  }

  // Sélection
  if( b = = 3 )
    cout << "\n B est 3 ";

  return 0;
}
```

Si une ou plusieurs instructions ne doivent être exécutées que dans une situation particulière, le if (SI) est la structure appropriée. Par exemple afficher le résultat si une solution a été trouvée, jouer une valeur si c'est dans l'intervalle réglementaire.

Le **if** est la structure de décision de base, elle détermine si les instructions contraintes entre les accolades seront exécutées ou non. **Si** la condition globale placée dans la parenthèse est vraie, les instructions sont exécutées une fois, **sinon**, on saute par-dessus le bloc qui ne sera jamais exécuté pour continuer l'exécution séquentielle du programme.

Les **if** sont utilisée pour les **comparaisons** :
- ? Lors de recherche d'éléments parmi un ensemble
- ? Lors de tris de données

Ils sont utilisés pour **tester un drapeau** :
- ? Après une boucle pour vérifier la cause de l'arrêt
- ? Après une recherche, pour vérifier si la clé a été trouvée

Servent à décider le **déclenchement d'une action** :
- ? Action dépendante d'une quantité (swapper quand la taille des données est trop grande)
- ? Action dépendante du choix dans un menu
- ? Sortie du programme lorsqu'une action a échoué. Le if teste la valeur de retour d'une fonction.

Ils servent à **initialiser des variables** selon d'autres variables ou conditions
- ? Mémoriser une option choisie via l'interface
- ? Fixer une option selon des paramètres du système comme la résolution par exemple.

Ils servent à **sélectionner certaines opérations** en fonction des options initiales
- ? Affichage 2D ou 3D selon l'option.
- ? Ajustement des calculs selon que l'usager veut une interpolation ou non.
- ? Choisir un module du programme en fonction des caractéristiques de l'usager comme l'âge.

Etc.

On remarque la dépendance dans les exemples ci-dessus avec les expressions **en fonction de**, **selon, choisi, dépendant de, vérifier.**

La structure de sélection if-else

if (condition **)** **{** bloc d'instructions dépendant de la condition; **}** **else** **{** bloc d'instructions alternatif; **}**

Condition testée avant d'exécuter le bloc d'instruction, si la condition est vraie, exécution du bloc dépendant, sinon exécution du bloc du else

```
int main(void)
{
  int a = 3;
  int b = 5;

 // Sélection
  if( a = = 3 )
  {
    cout << "\n A est 3 ";
  }
  else
  {
    cout << "\n A n'est pas 3 ";
  }

  // Sélection
  if( b = = 3 )
    cout << "\n B est 3 ";
  else
    cout << "\n B n'est pas 3 ";

  return 0;
}
```

```
A est 3
B n'est pas 3
```

Si une circonstance doit décider laquelle de deux séries d'instructions doit être exécutée, le if-else (SI-SINON) est la structure appropriée. Par exemple pour choisir entre deux sorties écrans, selon que tout va bien ou que rien ne va.

Le **if** est la structure de décision de base, elle détermine si les instructions contraintes entre les accolades subséquentes seront exécutées ou non. **Si** la condition globale placée dans la parenthèse est vraie, les instructions sont exécutées une fois, **sinon**, on saute par-dessus le bloc du if pour exécuter celui du else. Exactement un des deux blocs sera exécuté, et ceci, une seule fois.

Les **if-else** sont utilisée pour la **gestion d'erreurs** :

? Si l'erreur survient, terminer le programme, sinon continuer.

Ils sont utilisés pour **tester un drapeau** :

? Après une boucle pour vérifier la cause de l'arrêt et choisir la sortie écran en conséquence.

? Après une recherche, pour vérifier si la clé a été trouvée et sélectionner le bon affichage.

Ils servent à **minimiser le nombre de tests if** (temps d'exécution)

? Lorsque un if est vrai, l'autre if avec une condition qui ne recoupe pas, ne devrait pas être testé. Il convient donc de le placer dans le else même si la condition n'est pas parfaitement inverse (donc faire le test dans le else) Par exemple

```
if( a < 100 )
  …
else if( a == 200 )
  …
```

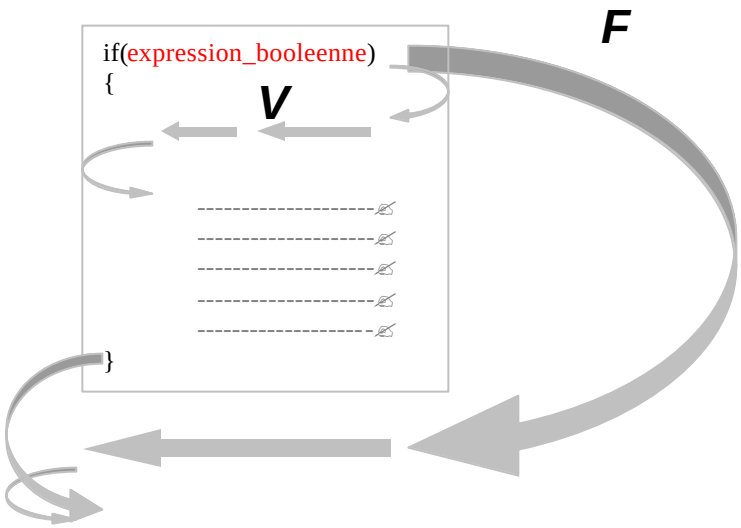
Ils servent aussi à **sélectionner certaines opérations** en fonction des options initiales

? Affichage 2D ou 3D selon l'option, toujours choix entre deux options

Etc.

On remarque la dépendance dans les exemples ci-dessus avec les expressions **en fonction de**, **selon, choisi, dépendant de, vérifier.**

Voici une illustration du flux d'exécution à la rencontre de l'expression booléenne contenue dans un **if**. Si la condition est VRAI, le bloc d'instructions sera exécuté. Si l'expression booléenne s'évalue à FAUX, alors l'exécution suivra la flèche F et ira se positionner après le bloc d'instruction du if.



Dans le cas d'un if/else, il y aura exécution d'un des deux blocs, certainement un, mais un seul. Ce sera soit celui du **if**, soit celui du **else**. Si l'expression booléenne est VRAI, le bloc du if sera exécuté mais celui du else sera ignoré. Si l'expression booléenne évalue à FAUX, le bloc du if sera ignoré mais le bloc du else sera exécuté.

```
if( expression_booleenne )
{
        V
}
else
{
        F
}
```

Exercices

1. Est-ce que ces deux bouts de code sont équivalents ? Si non, expliquer dans quelle situation donnent-elles un résultat équivalent, Et dans quelle situation elles ne le font pas.

```
if(test == 1)
        cout << ?symmetric?;
else
        cout << ?asymmetric?;


if(test == 1)
        cout << ?symmetric?;
if(test == 0)
        cout << ?asymmetric?;
```

2. Voici un pseudocode, écrire le code en c ou en c++ correspondant

Déclaration d'une variable entière pour l'âge

Affichage d'un message de bienvenue et de l'indication d'entrer l'âge au clavier

Lecture de l'âge entré au clavier

SI l'âge entré au clavier est supérieur ou égal à 18 ans
    Afficher que la personne est majeure

SINON
    Afficher que la personne est mineure

Voici des exemples d'exécution

MS-Dos ou Console
Programme de verification de la majorite Entrez votre age : 17 La personne est mineure

MS-Dos ou Console
Programme de verification de la majorite Entrez votre age : 18 La personne est majeure

Une solution en C++

```
int age;
cout << ?Programme de verification de la majorite\n?;
cout << ?Entrez votre age :  ?;
cin >> age;

if(age >= 18 )
        cout << ?La personne est majeure?;
else
        cout << ?La personne est mineure?;
```

Une solution en C

```
int age;
printf(?Programme de verification de la majorite\n?);
printf(?Entrez votre age :  ?);
scanf(&age);

if(age >= 18 )
        printf( ?La personne est majeure?);
else
        printf(?La personne est mineure?);
```

3. Voici un pseudocode, écrire le code en c ou en c++ correspondant

Programme d'affichage de la météo

Déclaration d'une variable pour retenir le choix de l'usager

Affichage d'un menu et de l'indication de choisir un élément de météo

Lecture du choix

SI le choix est ensoleillé
    Afficher un soleil
SINON SI le choix est pluvieux
    Afficher de la pluie
SINON SI le choix est un ciel étoilé
    Afficher des étoiles
SINON
    Afficher une erreur

Voici des exemples d'exécution

MS-Dos ou Console
Choisissez la meteo a afficher : 1. Ensoleille 2. Pluvieux 3. Ciel etoile 1 (((O)))

MS-Dos ou Console
Choisissez la meteo a afficher : 1. Ensoleille 2. Pluvieux 3. Ciel etoile 2 |||||||||

MS-Dos ou Console
Choisissez la meteo a afficher : 1. Ensoleille 2. Pluvieux 3. Ciel etoile 3 ****

Une solution en C++ :

```
int choix;
cout << ?Choisissez la meteo a afficher\n?;
cout << ?1. Ensoleille\n2. Pluvieux\n3. Ciel
        etoile\n\n?;
cin >> choix;
cout << ?\n?;

if(choix == 1 )
        cout << ?(((O)))?;
else if(choix == 2)
        cout << ?|||||||||||||?;
else if(choix == 3)
        cout << ?****?;
else
        cout << ?Choix invalide?;
```

Une solution en C:

```
int choix;
printf( ?Choisissez la meteo a afficher\n?);
printf( ?1. Ensoleille\n2. Pluvieux\n3. Ciel
          etoile\n\n?);
scanf(&choix);
printf(?\n?);

if(choix == 1 )
        printf(?(((O)))?);
else if(choix == 2)
        printf(?|||||||||||||?);
else if(choix == 3)
        printf(?****?);
else
        printf(?Choix inva
```

```
Choix    1    2
3
 ?       V    F
F

 ?       F    V
F
```

4. Voici un pseudocode, écrire le code en c ou en c++ correspondant

Programme de sélection des participants dans un endroit miniature

Déclaration d'une variable pour mémoriser l'entrée de l'usager

Affichage d'un message explicatif

Lecture de l'entrée

SI le choix plus grand ou égal à 150
    Interdiction d'entrer dans le jardin pour enfant
SINON SI le choix est plus petit que 150 mais plus grand uqe zéro
    On peut entrer dans le jardin
SINON
    Afficher une erreur

Voici des exemples d'exécution :

MS-Dos ou Console
Bienvenue au jardin d'Alice ! Grandeur de l'enfant: 144 Porte ouverte

MS-Dos ou Console
Bienvenue au jardin d'Alice ! Grandeur de l'enfant: 167 Porte fermee

MS-Dos ou Console
Bienvenue au jardin d'Alice ! Grandeur de l'enfant: 0 Impossible !

Une solution :

```
int grandeur;
cout << ?Bienvenue au jardin d'Alice\n ?;
cout << ?Grandeur de l'enfant?;
cin >> grandeur;

if(grandeur >= 150 )
        cout << ?Porte fermee?;
else if(grandeur > 0)
        cout << ?Porte ouverte?;
else
        cout << ?Impossible !?;
```

Une solution :

```
int grandeur;
cout << ?Bienvenue au jardin d'Alice\n ?;
cout << ?Grandeur de l'enfant?;
cin >> grandeur;

if(grandeur < 150 && grandeur > 0)
        cout << ?Porte ouverte?;
else if(grandeur > 150 )
        cout << ?Porte fermee?;
else
        cout << ?Impossible !?;
```

Une solution :

```
int grandeur;
cout << ?Bienvenue au jardin
     d'Alice\n?;
cout << ?Grandeur de l'enfant?;
cin >> grandeur;

if(grandeur <= 0)
        cout << ?Impossible !?;
else if(grandeur > 150)
        cout << ?Porte fermee?;
else
        cout << ?Porte ouverte?;
```

Une solution :

```
int grandeur;
cout << ?Bienvenue au jardin
d'Alice\n?;
cout << ?Grandeur de l'enfant ?;
cin >> grandeur;

if(grandeur >= 150 )
        cout << ?Porte fermee?;
else if(grandeur <= 0)
        cout << ?Impossible !?;
else
        cout << ?Porte ouverte?;
```

Une solution :

```
int grandeur;
cout << ?Bienvenue au jardin
d'Alice\n ?;
cout << ?Grandeur de l'enfant?;
cin >> grandeur;

if(grandeur <= 0)
        cout << ?Impossible !?;
else if(grandeur < 150)
        cout << ?Porte ouverte?;
else
        cout << ?Porte fermee?;
```

5. Voici un pseudocode, écrire le code en c ou en c++ correspondant

Programme de sélection des pilotes d'un prototype d'avion

Déclaration d'une variable pour mémoriser l'entrée de l'usager

Affichage d'un message explicatif

Lecture de l'entrée

SI le choix plus grand que 185 ou plus petit que 175
    Interdiction de piloter
SINON
    On peut piloter

Voici des exemples d'exécution

MS-Dos ou Console
Sélection de pilote Grandeur : 180 Accepte

MS-Dos ou Console
Sélection de pilote Grandeur : 150 Refuse

MS-Dos ou Console
Sélection de pilote Grandeur : 200 Refuse

Une solution en C++

```
int grandeur;
cout << ?Selection de pilote\n?;
cout << ?Grandeur :  ?;
cin >> grandeur;

if(grandeur > 185 || grandeur < 175 )
        cout << ?Refuse?;
else
        cout << ?Accepte?;
```

Une solution en C++

```
int grandeur;
cout << ?Selection de pilote\n?;
cout << ?Grandeur :  ?;
cin >> grandeur;

if(grandeur <= 185 && grandeur >= 175 )
        cout << ?Accepte?;
else
        cout << ?Refuse?;
```

6. Programme qui demande la couleur d'une carte : rouge ou noire et qui affiche si on a gagné

Déclaration d'une variable pour retenir le choix de l'usager
Déclaration d'une variable pour choisir un nombre aléatoire
Déclaration d'une variable pour retenir la bonne couleur aléatoire

Appeler un nombre aléatoire
Utiliser le nombre aléatoire pour piger une couleur au hasard

Affichage d'un message de bienvenue et de l'indication de choisir rouge ou noir au clavier

Lecture de l'entrée clavier

SI l'entrée clavier est le résultat du tirage aléatoire
    Afficher que la partie est gagnée

SINON
    Afficher que la partie est perdue

Voici des exemples d'exécution

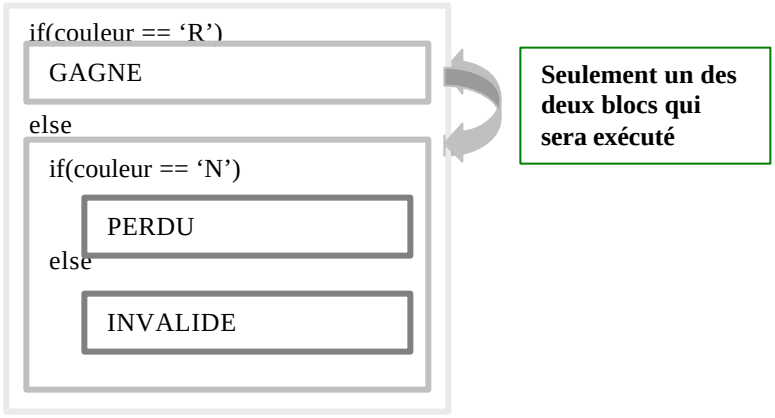
MS-Dos ou Console
Bienvenue au jeu rouge-noir ! Entrez votre choix (R/N) : R La partie est gagnee

MS-Dos ou Console
Bienvenue au jeu rouge-noir ! Entrez votre choix (R/N) : O La partie est perdue

Une solution en C++

```
char choix, couleur_secrete;
int aleatoire;

srand(time(0));
aleatoire = rand()%2;
if(aleatoire)
        couleur_secrete = 'R';
else
        couleur_secrete = 'N';

cout << ?Bienvenue au jeu rouge-noir !\nEntrez votre choix
(R/N) : ?;
cin >> choix;

if(choix == couleur_secrete )
        cout << ?La partie est gagnee?;
else
        cout << ?La partie est perdue?;
```

7. Écrire des versions équivalentes de ce code



Une solution en C++

```
char choix;

cout << ?Bienvenue au jeu rouge-noir !\nEntrez votre choix
(R/N) : ?;
cin >> choix;

if(choix == 'R')
        cout << ?La partie est gagnee?;
else
{
        if(choix == 'N'
                cout << ?La partie est perdue?;
        else
                cout << ?Invalide?;
}
```

Une solution en C++

```
char choix;

cout << ?Bienvenue au jeu rouge-noir ! ?;
cout << ?\nEntrez votre choix (R/N) : ?;
cin >> choix;

if(choix == 'R')
        cout << ?La partie est gagnee?;
else if(choix == 'N'
        cout << ?La partie est perdue?;
else
        cout << ?Invalide?;
```

Une solution en C++

```
char choix;

cout << ?Bienvenue au jeu rouge-noir ! ?;
cout << ?\nEntrez votre choix (R/N) : ?;
cin >> choix;

if(choix == 'R')
        cout << ?La partie est gagnee?;
if(choix == 'N'
        cout << ?La partie est perdue?;
if(choix != 'N' && choix != 'R')
        cout << ?Invalide?;
```

La structure de sélection switch-case

switch (variable **) {** case constante : instructions ; break; **}**

Variable comparée à chaque constante jusqu'à ce que la relation variable == constante soit vérifiée

Le **switch-case** est utilisé à la place de if-else cascadés lorsqu'il est requis de comparer **une seule variable** avec différentes **constantes**.

Il sert notamment à choisir entre les options d'un menu, à différencier entre différents symboles lors de traitement de texte ou encore à déterminer une action selon une position dans des intervalles réguliers comme dans l'exemple ci-dessous.

```
int main(void)
{
  int a;
  do
    cin >> a;
  while(a < 0);

  if( a < 50 )
  {
    cout << "\n A est entre 0 et 50 ";
  }
  else
  {
    if( a < 100 )
    {
      cout << "\n A est entre 50 et 100 ";
    }
    else
    {
      if(a < 150)
        cout << "\n A est entre 100 et 150";
      else
        cout << "\n A est superieur a 150";
    }
  }
  return 0;
}
```

⟺

```
int main(void)
{
  int a;
  do
    cin >> a;
  while(a < 0);
  a /= 50;

  switch( a )
  {
    case 0 : cout << "\n A est entre 0 et 50 ";
             break;
    case 1 : cout << "\n A est entre 50 et 100 ";
             break;
    case 2 : cout << "\n A est entre 100 et 150 ";
             break;
    default : cout << "\n A est superieur a 150 ";
  }

  return 0;
}
```

```
121
A est entre 100 et 150
```

```
121
A est entre 100 et 150
```

Exemple du jeu de cartes

```
cin >> valeur >> couleur;
cout << endl;

switch(valeur)
{
  case 1 :        cout << "As";          break;
  case 2 :
  case 3 :
  case 4 :
  case 5 :
  case 6 :
  case 7 :
  case 8 :
  case 9 :
  case 10 :       cout << valeur;        break;
  case 11 :       cout << "Valet";       break;
  case 12 :       cout << "Dame";        break;
  case 13 :       cout << "Roi";         break;
  default :       cout << "Erreur";      exit(1);
}

cout << " de ";

switch(couleur)
{
  case 'D' :      cout << "Carreau.";    break
  case 'C' :      cout << "Coeur";       break;
  case 'P' :      cout << "Pique";       break;
  case 'T' :      cout << "Trefle";      break;
  default :       cout << "Erreur";      exit(1);
}
```

4 C (entrée clavier)
4 de Cœur

12 T (entrée clavier)
Dame de Trefle

Il est à noter que un seul case peut-être vrai pour une exécution donnée du switch.

Cependant, à partir d'une comparaison réussie (valeur_du_switch == constante_d_un_case) l'exécution des instructions associées commence. Cette exécution ne se termine que lorsqu'il y a rencontre d'un break. Sans break les instructions des autres lignes sont exécutées à la suite et même celles des autres case.

C'est pourquoi dans l'exemple ci-contre, beaucoup de case n'ont pas d'instructions associées mais leur validité causera l'exécution de la ligne du **case 10 :** puisqu'il n'y a aucun break d'ici-là.

Par exemple, si valeur est égale à 4, l'exécution commence à **case 4 :** et continue jusqu'au premier break rencontré. Donc, les instructions de case 4 à case 10 inclusivement seront exécutées.

Exemple du menu

```
cout << "A. Jeu\n"
     << "B. Statistiques\n"
     << "C. Aide\n"
     << "D. Sortie\n";
cin >> choix;
switch(choix)
{
  case 'a' :
  case 'A' :      play();   break;
  case 'b' :
  case 'B' :      stat();   break;
  case 'c' :
  case 'C' :      aide();   break;
  case 'd' :
  case 'D' :      exit(0);
  default :       cin >> choix;
}
```

A. Jeu
B. Statistiques
C. Aide
D. Sortie
c (entrée clavier)

Qu'est-ce qu'une boucle ?

Dans le programme, s'il arrive que une même séquence d'opérations est exécutée plusieurs fois consécutives, à la place de l'écrire mille fois dans le corps du programme grâce aux boucle, il est possible de ne l'inscrire qu'une seule fois et d'indiquer le nombre d'exécution désiré.

```
int main(void)
{
  int x, y, z, w, a;
  x = 10; y = 22; z = 444; w = 3;

  a = 1;

  if(x > 0)
  {
    a  *= x;
    x--;
  }

  if(x > 0)
  {
    a  *= x;
    x--;
  }

  if(x > 0)
  {
    a  *= x;
    x--;
  }

  if(x > 0)
  {
    a  *= x;
    x--;
  }

if(x > 0)
  {
    a  *= x;
    x--;
  }

  /* Mais combien de fois ?? */

  printf("Le resultat est : %d", a);

  return 0;
}
```

```
int main(void)
{
  int x, y, z, w, a;
  x = 10; y = 22; z = 444; w = 3;

   a = 1;

   while( x > 0 )
   {
      a  *= x;
      x- -;
   }

  printf("Le resultat est : %d", a);

  return 0;
}
```

ou

```
int main(void)
{
  int x, y, z, w, a;
  y = 22; z = 444; w = 3;

   a = 1;

   for( x=10 ; x > 0; x- - )
      a  *= x;

  printf("Le resultat est : %d", a);

  return 0;
}
```

Ici, tant que X n'est pas 0, on répète l'opération de multiplication et on décrémente X.

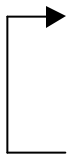
? X pourrait être très grand, trop grand, l'intérêt de la programmation est d'écrire une seule fois l'opération que la machine exécutera 1000 fois par exemple.

? X pourrait être inconnu, il pourrait dépendre de l'usager, une entrée au clavier qui décide le nombre de fois ou une entrée au clavier qui dit STOP (souvent la touche escape) , il pourrait aussi dépendre de la quantité de données à traiter, il pourrait dépendre de l'heure. Les boucles permettent de travailler TANT QUE cela est possible et nécessaire de le faire, sans savoir à l'avance la longueur du travail.

L'instruction de sélection (if, if/else, switch/case) déterminait si un bloc d'instruction allait être exécuté 0 ou 1 fois, les boucles elles, déterminent si un bloc d'instructions sera exécuté 0 ou 1 ou 2 ou 3 ou X fois.

Pour les programmeurs qui connaissent l'instruction **goto**, ou encore le saut **(jmp)**, la boucle peut être comparé à un bloc d'instruction rattaché à un if, mais qui se terminerait par un saut vers la ligne du if, pour pouvoir recommencer tant que la condition est vraie.

```
if(condition)
{
  // bloc exécuté 0 ou 1 fois

}
```



```
if(condition)
{


}
```

?

```
while(condition)
{
   // bloc exécuté 0 ou 1
   // ou 2 ou  3 ou 4  ou X fois

}
```

Une autre façon de comprendre les boucles est de les dérouler, c'est à dire d'écrire la suite d'instructions séquentielle équivalente.

```
for(  i = 1;  i<10; i++  )
{
  cout << "Allo " << i << " fois.\n";
}
```

?

```
i = 1;
cout << "Allo " << i << " fois.\n"; i++;
cout << "Allo " << i << " fois.\n"; i++;
cout << "Allo " << i << " fois.\n"; i++;
cout << "Allo " << i << " fois.\n"; i++;
cout << "Allo " << i << " fois.\n"; i++;
cout << "Allo " << i << " fois.\n"; i++;
cout << "Allo " << i << " fois.\n"; i++;
cout << "Allo " << i << " fois.\n"; i++;
cout << "Allo " << i << " fois.\n"; i++;
```

?

?

```
i = 1;
cout << "Allo " << i << " fois.\n"; i=2;
cout << "Allo " << i << " fois.\n"; i=3;
cout << "Allo " << i << " fois.\n"; i=4;
cout << "Allo " << i << " fois.\n"; i=5;
cout << "Allo " << i << " fois.\n"; i=6;
cout << "Allo " << i << " fois.\n"; i=7;
cout << "Allo " << i << " fois.\n"; i=8;
cout << "Allo " << i << " fois.\n"; i=9;
cout << "Allo " << i << " fois.\n"; i=10;
```

?

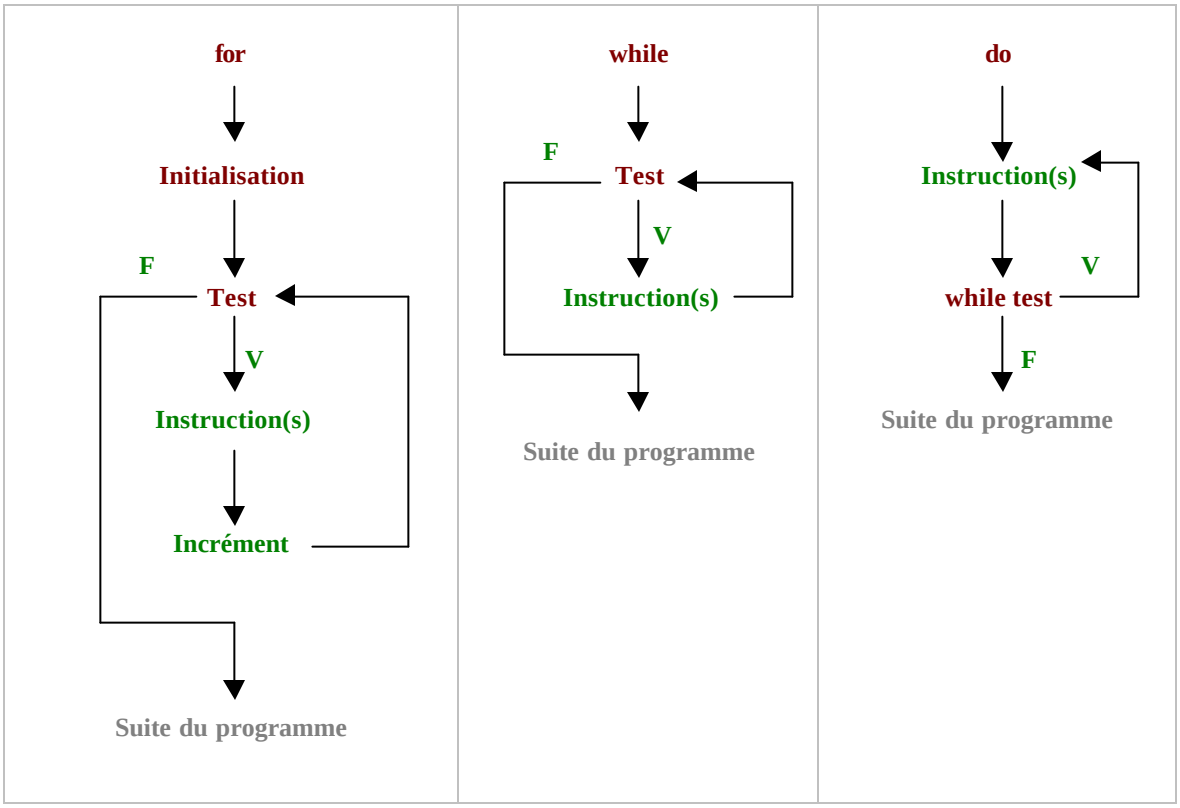
```
cout << "Allo " << 1 << " fois.\n";
cout << "Allo " << 2 << " fois.\n";
cout << "Allo " << 3 << " fois.\n";
cout << "Allo " << 4 << " fois.\n";
cout << "Allo " << 5 << " fois.\n";
cout << "Allo " << 6 << " fois.\n";
cout << "Allo " << 7 << " fois.\n";
cout << "Allo " << 8 << " fois.\n";
cout << "Allo " << 9 << " fois.\n";
i = 10;
```

```
Allo 1 fois.
Allo 2 fois.
Allo 3 fois.
Allo 4 fois.
Allo 5 fois.
Allo 6 fois.
Allo 7 fois.
Allo 8 fois.
Allo 9 fois.
```

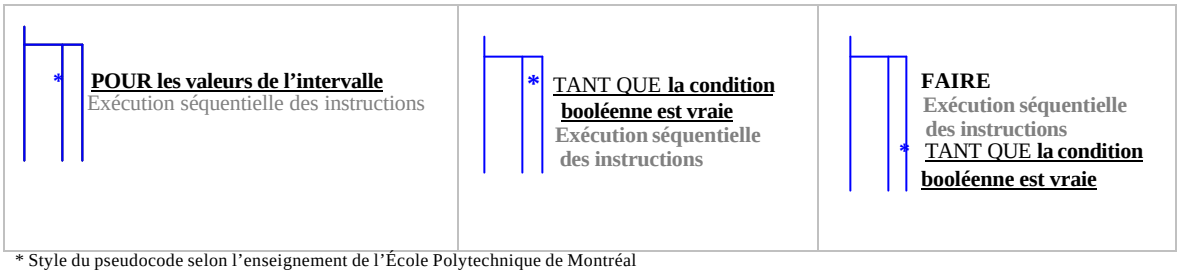
Trois sortes de boucle (structure de répétition)

Boucle for	***Boucle while***	***Boucle do-while***
Nombre de tours connu à l'avance	Nombre total de tours inconnu. Peut-être zero.	Au moins une exécution du corps de la boucle. Nombre total de tours inconnu.
for(initialisation; condition; increment) *{* *instructions;* *}*	*while(condition)* *{* *instructions;* *}*	*do* *{* *instructions;* *}while(condition);*

Diagrammes de flux (flow-diagrams)

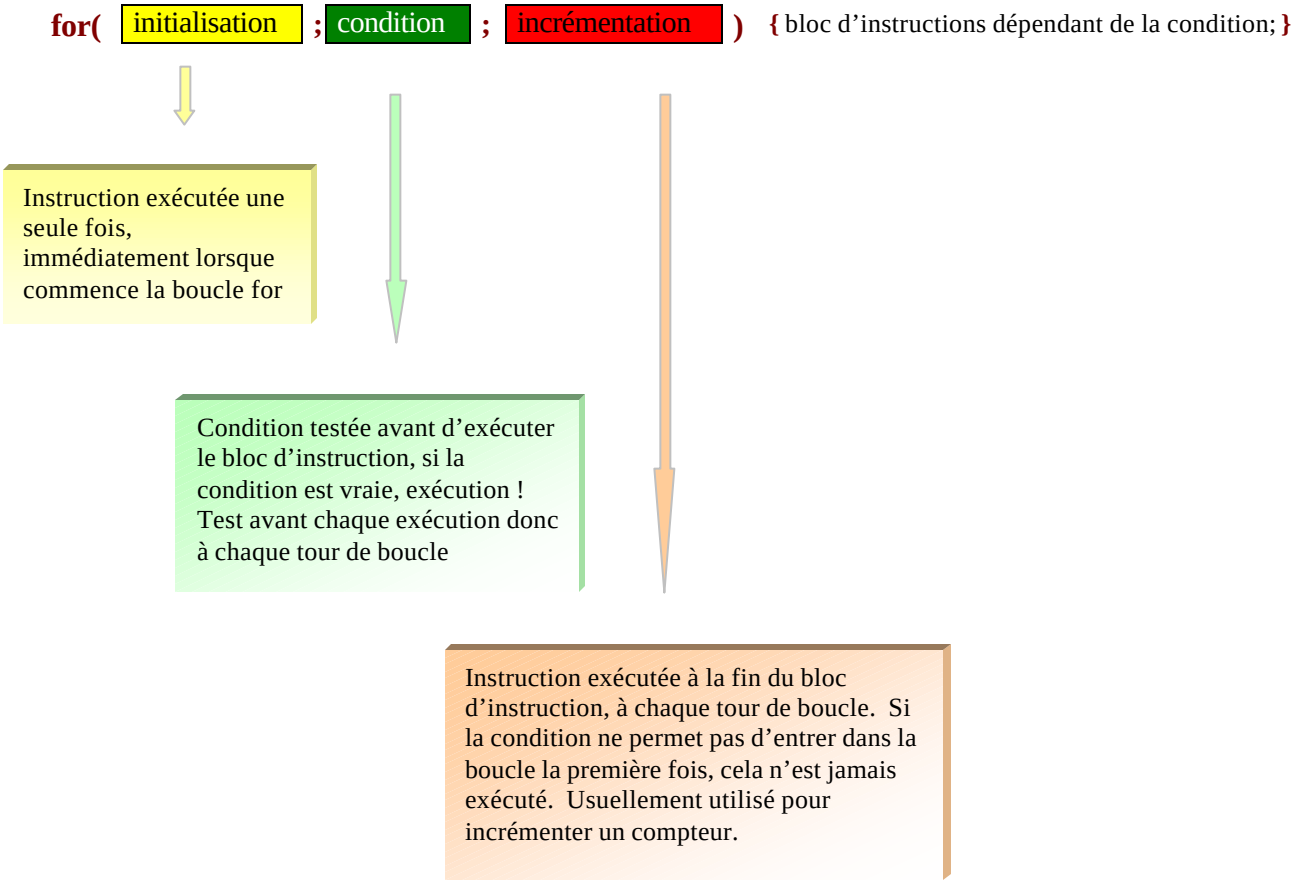


Pseudocode



* Style du pseudocode selon l'enseignement de l'École Polytechnique de Montréal

La boucle for



Une boucle for est utilisée pour une opération à répéter un **nombre prédéterminé de fois**. La boucle for(i=1; i<10; i++) {} se lit **POUR** les valeurs de 1 à 9, exécuter …

Par exemple,

? donner le même traitement **pour toutes** les cases d'un tableau de nombres
Une simple boucle for sert à initialiser toutes les cases à zéro
C'est très utilisé dans le traitement de tableaux multidimensionnel, où l'on rencontre des boucles for imbriquées pour manipuler des représentations de matrices.

? tester **tous les** nombres entre 1 et une certaine limite, par exemple pour la recherche de facteurs, de nombres premiers.

? entrer dans le programme un **nombre prédéterminé** de données

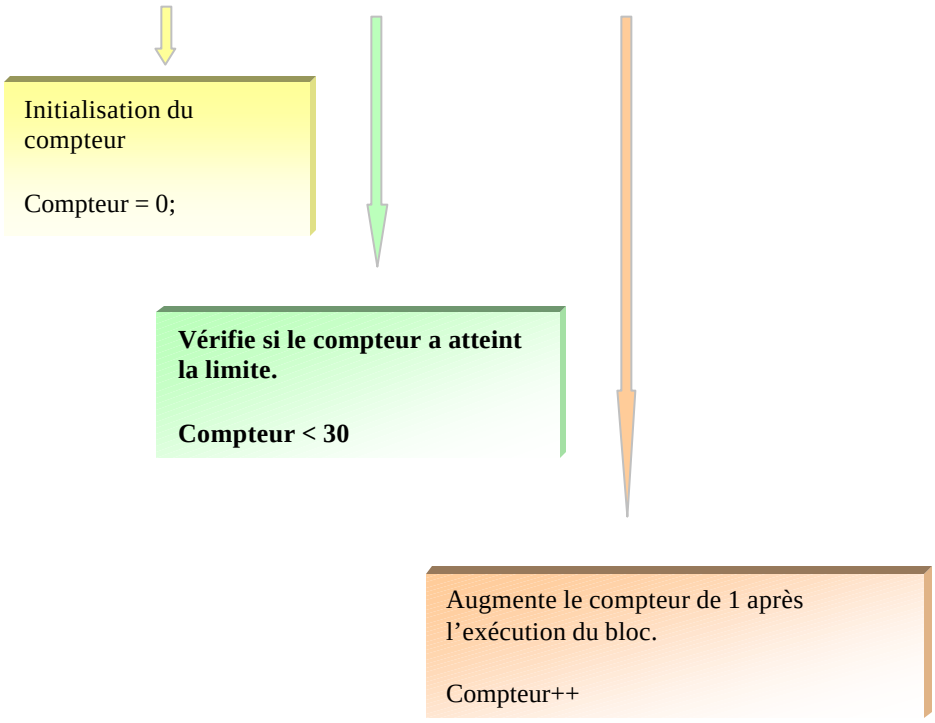
Ne jamais oublier que l'incrémentation est exécutée après le bloc d'instruction (donc juste avant le test de la condition pour le prochain tour). Ainsi, après la terminaison de la boucle, nous savons que l'incrémentation a eu lieu une fois de plus que nécessaire, avant que la condition ne soit falsifiée.

for(i=0; **i<=10**; i++) {}
Cette boucle se termine avec **i = 11**

Mots-clé : **pour, tous/toutes, nombre prédéterminé**

La boucle for – Usage classique

for(initialisation **;** condition **;** incrémentation **)** **{** bloc d'instructions dépendant de la condition; **}**




```
// Ce programme calcule la moyenne de nombres

int main(void)
{
   int n, moyenne;

   cout << "Entrez les 5 nombres";

  // Boucle de zero à quatre
   for(int c=0; c<5; c++)
   {
      cin >> n;
      moyenne += n;
   }

   moyenne /=  5;

   cout << "\nLe résultat est " << moyenne
         << endl << endl;

   return 0;
}
```

Étudions en détail la syntaxe et la sémantique de la boucle for. Nous voyons l'utilisation la plus courante de chacun des champs, mais aussi sa signification véritable.

```
for(            ;           ;           )
{
        Bloc
        d'instructions

}
```

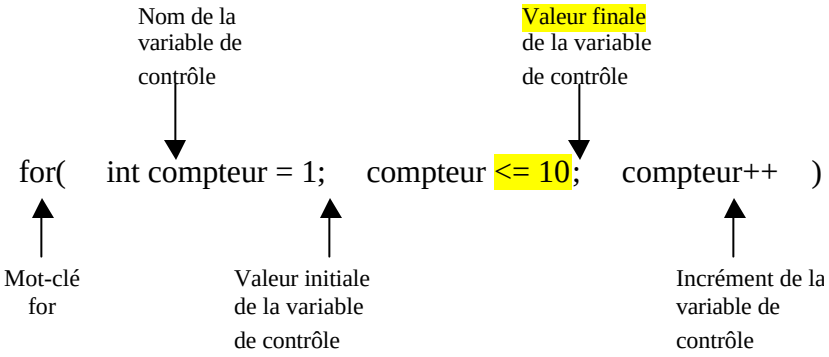
La boucle for se constitue de la parenthèse de contrôle et du bloc d'instructions entre accolades. La parenthèse de contrôle se divise en 3 parties séparées par des point-virgule.

```
for(  Initialisation  ;  Condition  ;  Instruction de fin de tour
                                       Incrémentation  )
{
        Bloc
        d'instructions

}
```

Les trois parties de la parenthèse de contrôle sont respectivement utilisées pour l'initialisation, la condition de continuation et l'instruction d'incrémentation. En fait, la mécanique du for prédispose cet usage; la partie initialisation est exécutée une seule fois avant la première itération, la première exécution du bloc d'instruction. La partie condition est testée avant chaque exécution du bloc d'instructions. L'instruction de fin de tour est exécutée après chaque exécution du bloc d'instruction, même après le dernier tour.

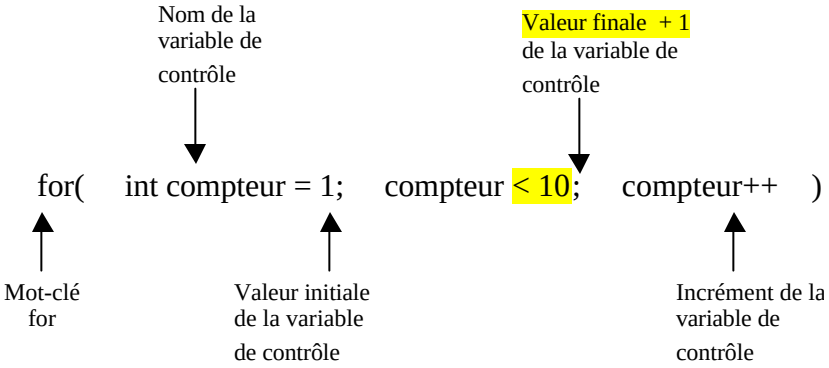
```
for(  int i = 0  ;  i <= 10  ;  i++  )
{
        Bloc
        d'instructions

}
```

Voici un exemple classique de code pour une boucle qui fait 10 itérations.



Ici les valeurs de compteur s'échelonneront, par incréments de 1, de 1 à 10 inclusivement. À la sortie de la boucle, compteur sera 11 et le test sera faux car 11 est supérieur à 10 (rappelons que l'opérateur <= retourne vrai pour l'égalité, donc permet un tour de boucle même quand compteur est 10.)

En C++ mais pas en C, l'espace d'initialisation peut servir à déclarer une variable, étape habituellement utilisée pour la création et l'initialisation d'un compteur. Le domaine de validité de l'identifiant est la boucle for uniquement, il ne sera pas possible à la sortie de la boucle de manipuler ou afficher la variable *compteur.*



Ici les valeurs de compteur s'échelonneront, par incréments de 1, de 1 à 9 inclusivement À la sortie de la boucle, compteur sera 10 et le test sera faux car le test requiert plus petit que 10 pour l'exécution (rappelons que l'opérateur < retourne faux pour l'égalité, ce qui empêche le corps de la boucle de s'exécuter lorsque le compteur est à 10.)

Dans certains compilateurs, le nom de la variable utilisée dans la boucle précédente ne peut être redéclaré dans une boucle ultérieure, dans certains compilateurs, probablement qu'elle est considérée comme statique.

Il arrive que deux boucles for soient imbriquées l'une dans l'autre, on les appelle aussi cascadées. La boucle interne est totalement dépendante de la boucle externe pour son exécution. Si la boucle externe ne fait aucune itération, la boucle interne n'est pas exécutée, si la boucle externe fait une itération, la boucle interne fait toutes les siennes une fois, si la boucle externe fait deux itérations, la boucle interne fait 2 cycles complets.

Compteur1	Compteur2
0	0
0	1
0	2
0	bloquée
1	0
1	1
1	2
1	bloquée
2	0
2	1
2	2
2	bloquée
3	0
3	1
3	2
3	bloquée
4	0
4	1
4	2
4	bloquee
bloquee	

```
for(   compteur1 = 0;    compteur1 < 5;    compteur1++    )
{

    for(   compteur2 = 0;    compteur2 < 3;  compteur2++    )
     {

          cout << compteur1 << ' ' << compteur2 << endl;

      }

 }
```

Comme les niveaux d'un boulier, compteur2 passera par toutes ses valeurs de 0 à 2 pour chaque valeur de compteur1. Toutes les combinaisons sont donc utilisées. La boucle interne s'exécute 5x3 = 15 fois Ceci est utile pour

? **de la combinatoire**
? **le parcours de toutes les cases d'un tableau 2D.**

Le cas présenté ci-haut est très simple, la valeur du compteur de la boucle externe n'influence pas le comportement de la boucle interne, cela décide seulement si l'exécution aura lieu. Il arrive parfois que la valeur du compteur de la boucle externe serve de limite pour le compteur de la boucle interne. La dépendance n'est plus seulement dans l'exécution de la boucle interne mais aussi dans son nombre d'itérations qui varie selon la progression de la boucle externe. Cette technique est utilisée :

? pour les tris afin de ne pas remanipuler une zone déjà triée
? dans les manipulations triangulaires, affichage de triangles, manipulation de parties de matrice.

Compteur1	Compteur2
0	bloquée
1	0
1	bloquée
2	0
2	1
2	bloquée
3	0
3	1
3	2
3	bloquée
4	0
4	1
4	2
4	3
4	bloquée
5	0
5	1
5	2
5	3
5	4
bloquée	

```
for(   compteur1 = 0;    compteur1 < 5;    compteur1++    )
{
   for( compteur2 = 0;  compteur2 < compteur1;  compteur2++ )
    {

         cout << compteur1 << ' ' << compteur2 << endl;

     }
 }
```

Ici, le nombre de tours de la boucle interne augmente progressivement avec l'indice de la boucle externe. D'autres relations peuvent être utilisées tel prendre le compteur externe comme valeur initiale

```
for( c2 = c1; c2 < 50; c2++ )
```

ou encore utiliser comme nombre d'itérations une limite moins le compteur externe.

```
for( c2 = 0; c2 < 50 - c1; c2++ )
```

L'affichage : exercices sur les boucles imbriquées

Exemple d'un compteur décimal

La solution impliquera 2 boucles imbriquées : L'une pour le premier facteur et l'autre pour le second.

```
00
01
02
03
04
…
```

Exemple de l'affichage des tables de multiplication

Essayez d'utiliser la fonction setw( ) pour aligner les affichages.

```
0x0=0  0x1=0  0x2=0 0x3=0  0x4=0    0x5=0
1x0=0  1x1=1  1x2=2 1x3=3  1x4=4    1x5=5
2x0=0  2x1=2  2x2=4 2x3=6  2x4=8    2x5=10
3x0=0  3x1=3  3x2=6 3x3=9  3x4=12   3x5=15
                …
```

Exemple du dessin d'un triangle

Écrire le code en fonction d'une taille de triangle définie par une variable.

La variable entière largeur contient à priori le nombre d'étoiles de la rangée du bas qui est aussi le nombre de lignes.

```
*
**
***
****
*****
******
```

Exemple du dessin d'une pyramide

Écrire le code en fonction d'une taille de triangle définie par une variable.

La variable entière largeur contient à priori le nombre de lignes.

```
     *
    ***
   *****
  *******
 *********
***********
```

Les sommations

Calcul d'une sommation $\sum_{i=0}^{5} (i^2)$

Indice : pour faire une sommation avec une boucle, il faut utiliser une variable qui sert d'accumulateur.

Important : ne pas oublier d'initialiser cette variable à zéro avant de commencer.

n	n*n	Somme
0	0	0
1	1	1
2	4	5
3	9	14
4	16	30
5	25	55
6	Boucle bloquée	

n	n*n	Somme
5	25	25
4	16	41
3	9	50
2	4	54
1	1	55
0	0	55
-1	Boucle bloquée	

Calcul d'une sommation $\sum_{j=1}^{3} \sum_{i=0}^{5} (i-j)^2$

Pour une sommation à deux variables, nous utiliserons deux boucles imbriquées.

j	i	(i-j)*(i-j)	Somme
1	0	1	0
1	1	0	1
1	2	1	2
1	3	4	6
1	4	9	15
1	5	16	31
1	6	Boucle bloquée	
2	0	4	35
2	1	1	36
2	2	0	36
2	3	1	37
2	4	4	41
2	5	9	50
2	6	Boucle bloquée	
3	0	9	59
3	1	4	63
3	2	1	64
3	3	0	64
3	4	1	65
3	5	4	69
3	6	Boucle bloquée	

Combinaisons

Calcul d'une factorielle

Écrire le code en fonction d'un nombre donné dans la variable n.
Le tableau ci-contre indique le contenu des variables du programme tout au long de l'exécution si n commence avec la valeur 5.

n	factorielle
5	5
4	20
3	60
2	120
1	120
0	Boucle bloquée

Vérifier si un nombre est premier

Écrire le code en fonction d'un nombre donné dans la variable nombre.

nombre	facteur	n/f == 0 ?	premier
25	2	F	1
25	3	F	1
25	4	F	1
25	5	V	0
25	6	F	0
25	7	F	0
25	8	F	0
25	9	F	0
25	10	F	0
25	11	F	0
25	12	F	0
25	25	Boucle non-exécutée	

Trouver et afficher les nombres premiers entre 2 et 100

Rappel de la définition d'un nombre premier : Un nombre est premier s'il n'a pas d'autre diviseurs que 1 et lui-même.

Autres indications : pour vérifier qu'une division s'effectue sans reste, on utilise l'opérateur modulo (%) qui calcule le reste d'une division entière à condition d'opérer sur des variables entières. On applique l'opérateur modulo et on vérifie ensuite si le résultat est égal à zéro.

nombre	facteur	n/f == 0 ?	premier
11	2	F	1
11	3	F	1
11	4	F	1
11	5	F	1
11	6	Boucle non-exécutée	

nombre	facteur	n/f == 0 ?	premier
17	2	F	1
17	3	F	1
17	4	F	1
17	5	F	1
17	6	F	1
17	7	F	1
17	8	F	1
17	6	Boucle non-exécutée	

Solutions proposées pour les exercices sur l'affichage

Exemple d'un compteur décimal

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

```
00
01
02
03
04
…
```

Exemple de l'affichage des tables de multiplication

```
for( int i=0; i<=10; i++ )
{
  for( int j=0; j<=5; j++ )
    cout << i << 'x' << j << '=' << i*j << '  ';
  cout << endl;
}
```

```
0x0=0  0x1=0  0x2=0 0x3=0  0x4=0   0x5=0
1x0=0  1x1=1  1x2=2 1x3=3  1x4=4   1x5=5
2x0=0  2x1=2  2x2=4 2x3=6  2x4=8   2x5=10
3x0=0  3x1=3  3x2=6 3x3=9  3x4=12  3x5=15
                …
```

Exemple du dessin d'un triangle

```
for( int ligne=1; ligne<=limite; ligne++ )
{
  for( int x=1;  x <= ligne;  x++ )
    cout << '*';
  cout << endl;
}
```

```
*
**
***
****
*****
******
```

Exemple du dessin d'une pyramide

```
for( int ligne=1; ligne<=limite; ligne++ )
{
  for( int x=1; x <= limite - ligne;  x++ )
    cout << ' ';
  for( int x=1; x <= 2*ligne-1;  x++ )
    cout << '*';
  cout << endl;
}
```

```
     *
    ***
   *****
  *******
 *********
***********
```

Solutions proposées pour les exercices de sommation

Calcul d'une sommation $\sum_{i=0}^{5} (i^2)$

```
#define DATA 5

int somme = 0;
for( int n=0; n<=DATA; n++ )
{
  somme += n*n;
}
```

n	n*n	Somme
0	0	0
1	1	1
2	4	5
3	9	14
4	16	30
5	25	55
6	Boucle bloquée	

```
#define DATA 5

int somme = 0;
for( int n=DATA; n>=0; n-- )
{
  somme += n*n;
}
```

n	n*n	Somme
5	25	25
4	16	41
3	9	50
2	4	54
1	1	55
0	0	55
-1	Boucle bloquée	

Calcul d'une sommation $\sum_{j=1}^{3} \sum_{i=0}^{5} (i-j)^2$

```
int somme = 0;

for( int j=1; j<=3; j++ )
  for( int i=0; i<=5; i++ )
    somme += (i-j) * (i-j);
```

j	i	(i-j)*(i-j)	Somme
1	0	1	0
1	1	0	1
1	2	1	2
1	3	4	6
1	4	9	15
1	5	16	31
1	6	Boucle bloquée	
2	0	4	35
2	1	1	36
2	2	0	36
2	3	1	37
2	4	4	41
2	5	9	50
2	6	Boucle bloquée	
3	0	9	59
3	1	4	63
3	2	1	64
3	3	0	64
3	4	1	65
3	5	4	69
3	6	Boucle bloquée	

Solutions proposées pour les exercices de combinaisons

Calcul d'une factorielle

```
#define DATA 5

int factorielle = 1;
for( int n=DATA; n>0; n-- )
{
  factorielle *= n;
}
```

n	factorielle
5	5
4	20
3	60
2	120
1	120
0	Boucle bloquée

Vérifier si un nombre est premier

```
int premier = 1;  int nombre;  cin >> nombre;

for( int facteur=2; facteur<nombre/2;
facteur++ )
{
  if(nombre% facteur = = 0)
     premier = 0;
}
```

nombre	facteur	n/f == 0 ?	premier
25	2	F	1
25	3	F	1
25	4	F	1
25	5	V	0
25	6	F	0
25	7	F	0
25	8	F	0
25	9	F	0
25	10	F	0
25	11	F	0
25	12	F	0
25	25	Boucle non-exécutée	

Trouver et afficher les nombres premiers entre 2 et 100

```
for( int n=2; n<=100; n++)
{
  for( int facteur=2, premier=1;
                  facteur<nombre/2; facteur++ )
  {
    if(nombre%facteur = = 0)
      premier = 0;
  }
  if(premier) cout << n << ' ';
}
```

nombre	facteur	n/f == 0 ?	premier
11	2	F	1
11	3	F	1
11	4	F	1
11	5	F	1
11	6	Boucle non-exécutée	

nombre	facteur	n/f == 0 ?	premier
17	2	F	1
17	3	F	1
17	4	F	1
17	5	F	1
17	6	F	1
17	7	F	1
17	8	F	1
17	6	Boucle non-exécutée	

Boucle while controlée par une sentinelle

Initialisation de la sentinelle

Sentinelle = 0;

initialisation ;

La boucle while est normalement utilisée pour répéter une suite d'instructions un **nombre indéterminé de fois**, selon une condition qui demande une information au moment de l'exécution.

Cela peut dépendre d'une entrée de données par l'usager ou encore des valeurs des données manipulées.

Mots-clés : **répéter, tant que, nombre indéterminé de fois**

while(condition **)** **{** bloc d'instructions dépendant de la condition; falsifie la condition ; **}**

Vérifie si la sentinelle a la valeur de sortie prédéfinie

Sentinelle != -1

Rend la sentinelle fausse dépendant du contexte, si une telle instruction n'est pas présente : on a une boucle infinie, ce qui est habituellement indésirable

```
scanf("%d", &Sentinelle );  /* lecture clavier en c */
cin >> Sentinelle;              // lecture clavier en c++
if(autre_condition) Sentinelle = -1;
```

```
// Ce programme calcule la moyenne de nombres

int main(void)
{
  int n, moyenne, compteur = 0;

  cout << "Entrez des nombres, -1 pour terminer";
  cin >> n;

 // Boucle au nombre de tours indéterminé
  while(n != -1)
  {
    moyenne += note;
    compteur++;
    cin >> n;  // permet de falsifier la condition
  }

  moyenne /=  compteur;

  cout << "\nLe résultat est " << moyenne
       << endl << endl;

  return 0;
}
```

Ici par exemple, une sentinelle sera activée par l'usager pour indiquer la fin de la boucle. Un programme équivalent serait une boucle infinie terminée par un break. Ceci permet de terminer la boucle dans le milieu du bloc et pas seulement à la fin.

```
while( 1 )
{
  partie_du_programme;
  cout << "\nVoulez-vous continuer ? (O/N)" ;
  cin >> choix;
  if(choix != 'o' && choix != 'O')
    break;
  suite_du_programme;
}
```

Une boucle while est souvent utilisée pour englober tout un programme à répéter **tant que** l'usager le veut.

```
char reponse = 'o';
while( reponse == 'O' || reponse == 'o' )
{
  tout_le_programme;
  cout << "\nVoulez-vous recommencer ? (O/N)" ;
  cin >> reponse;
}
```

Exemple de la lecture d'un fichier

Ne pas faire immédiatement cet exercice qui nécessite aussi des connaissance en traitement de fichier : simplement apprécier la solution.

Exemple de l'entrée clavier tant que l'usager le veut

L'usager a été instruit d'entrer différents nombres destinés au traitement et d'entrer le chiffre –1 quand il désire terminer.
On doit faire la somme de ces nombres et aussi les compter afin de pouvoir calculer la moyenne..

```
5
6
876
34
-1
(fini)
```

Exemple de la manipulation d'une chaîne de caractères

Lorsque le main commence son exécution, il reçoit dans une variable tableau les deux chaînes de caractères, respectivement –s et -r.

Écrire un programme pour faire l'analyse de l'une de ces chaînes de caractère.
Indice : on peut utiliser une boucle sur les caractères d'une chaîne et un switch sur le caractère courant pour l'analyser.

J'aimerais que ces variables booléennes (ou entières): soit affectées par l'analyse.
bool recursive;
bool safe_mode;

L'idée est donc d'affecter nos drapeaux selon ce que l'usager a spécifié sur la ligne de commande.

```
Machine :prompt# ./ monprog –s –r
```

Exemple de parties dans un « bestof »

Un BestOf est une manière de décider du gagnant de plusieurs parties d'un jeu : le gagnant est celui qui obtient le premier une majorité de partie sur un nombre total impair.

Par exemple, on peut décider vouloir jouer un BestOf 7, et dans ce cas, le premier à obtenir 4 parties gagnées gagne le trophée.

Il y a deux concepts dans ce problème : on veut jouer des parties TANT QUE aucun des scores n'atteint le bestof que l'on veut.

On désire pour chaque partie (boucle imbriquée) jouer un tour TANT QUE il n'y a pas de gagnant ni de partie nulle.

Exemple du programme qui exécute jusqu'à ESC

On veut afficher le code ascii des caractères entrés TANT QUE la touche entrée n'est pas ESC.

À titre indicatif, le code ascii de la touche ESC est 27.

Un autre indice est la façon de déterminer le code ascii d'un caractère qui est simplement de faire un CAST. On fait un cast en utilisant le nom d'un autre type entre parenthèse juste devant la variable.

cout << (cast)ma_lettre

```
a
Le code ascii pour la touche a est 95
(espace)
Le code ascii pour la touche   est  32
(ESC)
```

Exemple pour comparer des chaînes

On veut déterminer si deux chaînes de même longueur sont identiques.

Le résultat du code est de fixer la valeur de la variable booléenne (ou entière) dont le nom est identique

Texte1	Texte2	compare
B	B	1
O	O	1
U	U	1
C	G	0
H	R	0
E	E	0
null	Boucle bloquée	

Exemple pour compter des occurrences de caractères

Je veux compter combien d'espaces il y a dans une chaîne de caractères.

Note : le code ascii de espace est 32.

chaine	espace
B	0
[ ]	1
U	1
[ ]	2
H	2
E	2
null	Boucle bloquée

Exemple de la lecture d'un fichier

```
while( !fic.eof() )
{
   fic >> var1 >> var2 >> var3;
   fic.getline(buffer, 30);
}
```

Exemple de l'entrée clavier tant que l'usager le veut

```
cin >> nombre;
while( nombre != -1 )
{
  somme += nombre;
  cin >> nombre;
}
```

```
5
6
876
34
-1
(fini)
```

Exemple de la manipulation d'une chaîne de caractères

```
while( texte[++indice] )
{
  switch( texte[indice] )
  {
     case '-' : option = 1; break;
     case 's' : if(option) {option--; safe = 1;} break;
     case 'r' : if(option) {option--; recursive = 1;} break;
     default :
  }
}
```

```
Machine :prompt# ./ monprog –s –r
```

Exemple de parties dans un « bestof »

```
#define B_OF 7
while( aScore < (B_OF + 1)/2  && bScore < (B_OF + 1)/2  )
{

  while( !gagnant  && !partie_nulle  )
  {
    // jouer un tour
  }

  //  mettre à jour le score du gagnant
  if(gagnant == 1)
    aScore++;
  else if(gagnant == 2)
    bScore++;
}
```

Exemple du programme qui exécute jusqu'à ESC

```
char touche;
cin >> touche;
while( touche != 27 ) // tant que ce n'est pas ESC
{
   cout << endl << "Le code ascii pour la touche "
        << touche << " est " << (int)touche << endl;
  cin >> touche;
}
```

```
a
Le code ascii pour la touche a est 95
(espace)
Le code ascii pour la touche   est  32
(ESC)
```

Exemple pour comparer des chaînes

```
compare = 1;
while( texte1[++indice] && texte2[indice]  )
{
  if(texte1[indice] != texte2[indice])
     compare = 0;
}
```

Texte1	Texte2	compare
B	B	1
O	O	1
U	U	1
C	G	0
H	R	0
E	E	0
null	Boucle bloquée	

Exemple pour compter des occurrences de caractères

```
espace = 0; indice = -1
while( chaine[++indice] )
{
   if(chaine[indice] == 32)
     espace++;
}
```

chaine	espace
B	0
[ ]	1
U	1
[ ]	2
H	2
E	2
null	Boucle bloquée

Voir la section sur les tableaux et celle sur les listes chaînées pour plus d'exemples

Imiter une boucle for avec une boucle while (boucle while controlée par un compteur)

Initialisation du compteur

Compteur = 0;

initialisation ;

while(condition **)** **{** bloc d'instructions dépendant de la condition; incrémentation ; **}**

Vérifie si le compteur a atteint la limite.

Compteur < 30

Augmente le compteur de 1 après l'exécution du bloc.

Compteur++

```
// Ce programme calcule la moyenne de plusieurs notes

int main(void)
{
  int n, moyenne;

  cout << "Entrez 5 nombres.";

 // Boucle au nombre de tours déterminé (5)
  int c = 0;
  while(c < 5)
  {
    cin >> n;
    moyenne += n;
    c++;    // permet de falsifier la condition
  }

  moyenne /=  5;

  cout << "\nLe résultat est " << moyenne << endl << endl;

  return 0;
}
```

Pour bien comprendre le fonctionnement des boucles while et for, voici un exemple de boucle while qui imite l'exemple précédent de la boucle for.

Il est possible de faire le mimétisme de la boucle for en positionnant l'initialisation avant le mot **while**, en positionnant l'incrémentation après toutes les instructions du bloc et en testant pour le dépassement d'une limite dans la condition du while.

Ainsi, la boucle while est utilisée pour une suite de valeur connues, grâce à l'initialisation, au pas régulier et à la limite. Cette boucle peut donc être utilisée pour un nombre de tour prédéterminé même si c'est normalement l'usage des boucles for et non des boucles while.

Cet exemple montre le choix de certains programmeurs et aide également à comprendre la boucle for présentée ci-haut puisqu'elle lui est équivalente.

L'affichage

Exemple de l'affichage des tables de multiplication

```
int j, i = 0;
while(i<=10)
{
  j = 0;
  while(j<=5)
  {
    cout << i << 'x' << j << '=' << i*j << '   ';
    j++;
  }
  cout << endl;
  i++;
}
```

```
0x0=0  0x1=0  0x2=0 0x3=0  0x4=0  0x5=0
1x0=0  1x1=1  1x2=2 1x3=3  1x4=4  1x5=5
2x0=0  2x1=2  2x2=4 2x3=6  2x4=8  2x5=10
3x0=0  3x1=3  3x2=6 3x3=9  3x4=12  3x5=15
                    …
```

```
int j, i = -1;
while(++i <= 10)
{
  j = -1;
  while(++j<=5)
    cout << i << 'x' << j << '=' << i*j << '   ';
  cout << endl;
}
```

Exemple d'un compteur décimal

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

```
00
01
02
03
04
…
```

Exemple du dessin d'un triangle

```
int x, ligne = 1
while( ligne<=limite )
{
  x = 1;
  for(x<=ligne)
  {
    cout << '*';
    x++;
  }
  cout << endl;
  ligne++;
}
```

```
*
**
***
****
*****
******
```

Exemple du dessin d'une pyramide

```
int x, ligne = 1;
while(ligne<=limite)
{
  x = 0;  while( ++x <=  limite - ligne )
    cout << ' ';
  x = 0; while( ++x <= 2*ligne-1 )
    cout << '*';
  cout << endl;
  ligne++;
}
```

```
     *
    ***
   *****
  *******
 *********
***********
```

Combinaisons

Calcul d'une factorielle

```
#define DATA 5

int factorielle = 1;
int n = DATA;

while(n>0)
  factorielle *= n--;
```

n	factorielle
5	5
4	20
3	60
2	120
1	120
0	Boucle bloquée

Vérifier si un nombre est premier

```
int premier = 1; int facteur = 1;
int nombre;  cin >> nombre;
while( ++facteur < nombre/2 )
  if(nombre%facteur == 0)
    premier = 0;
```

```
int premier = 1; int facteur = 2;
int nombre;  cin >> nombre;
while(facteur < nombre/2 )
  if(nombre%facteur++ == 0)
    premier = 0;
```

nombre	facteur	n/f == 0 ?	premier
25	2	F	1
25	3	F	1
25	4	F	1
25	5	V	0
25	6	F	0
25	7	F	0
25	8	F	0
25	9	F	0
25	10	F	0
25	11	F	0
25	12	F	0
25	25	Boucle non-exécutée	

Trouver et afficher les nombres premiers entre 2 et 100

```
int premier, facteur, n=2;
while(n<=100)
{
  facteur = 2; premier = 1;
  while( facteur<nombre/2)
  {
    if(nombre/facteur == 0)
      premier = 0;
    facteur++;
  }
  if(premier) cout << n << ' ';
  n++;
}
```

```
int premier, facteur, n=1;
while(++n<=100)
{
  facteur = 1; premier = 1;
  while( ++facteur < nombre/2)
  {
    if(nombre/facteur == 0)
      premier = 0;
  }
  if(premier) cout << n << ' ';
}
```

Les sommations

Calcul d'une sommation $\sum_{i=0}^{5} (i^2)$

```
#define DATA 5
int somme = 0, n = 0;
while( n<=DATA )
{
  somme += n*n++;
}
```

n	n*n	Somme
0	0	0
1	1	1
2	4	5
3	9	14
4	16	30
5	25	55
6	Boucle bloquée	

ou

```
#define DATA 5
int somme = 0, n=DATA;
while( n>=0 )
{
  somme += n*n--;
}
```

n	n*n	Somme
5	25	25
4	16	41
3	9	50
2	4	54
1	1	55
0	0	55
-1	Boucle bloquée	

Calcul d'une sommation $\sum_{j=1}^{3} \sum_{i=0}^{5} (i-j)^2$

```
int i, j=1, somme = 0;
while( j<=3 )
{
  i = 0;
  while( i<=5 )
  {
    somme += (i-j) * (i - j);
    i++;
  }
  j++;
}
```

j	i	(i-j)*(i-j)	Somme
1	0	1	0
1	1	0	1
1	2	1	2
1	3	4	6
1	4	9	15
1	5	16	31
1	6	Boucle bloquée	
2	0	4	35
2	1	1	36
2	2	0	36
2	3	1	37
2	4	4	41
2	5	9	50
2	6	Boucle bloquée	
3	0	9	59
3	1	4	63
3	2	1	64
3	3	0	64
3	4	1	65
3	5	4	69
3	6	Boucles bloquée	

Boucle do-while controllée par une sentinelle

Initialisation de la sentinelle

Sentinelle = 0;

La boucle do-while est normalement utilisée pour répéter une suite d'instructions un **nombre indéterminé de fois** après une première exécution du bloc d'instructions, selon une condition qui demande une information au moment de l'exécution.

Mots-clés : **faire, répéter, tant que, nombre indéterminé de fois**

initialisation ;

do { bloc d'instructions dépendant de la condition; falsifie la condition ; } **while(** condition **);**

Rend la sentinelle fausse dépendant du contexte, si une telle instruction n'est pas présente : on a une boucle infinie, ce qui est habituellement indésirable

scanf("%d", &Sentinelle); ou
cin >> Sentinelle; ou
if(autre condition) Sentinelle = -1;

Vérifie si la sentinelle a la valeur de sortie prédéfinie

Sentinelle != -1

```
// Ce programme calcule la moyenne de nombres

int main(void)
{
  int n, moyenne, compteur = 0;

  cout << "Entrez des nombres, -1 pour terminer";

 // Boucle au nombre de tours indéterminé
  do {
    cin >> n;      // permet de falsifier la condition
    moyenne += note;
    compteur++;
  }
  while(n != -1);

  note++; // parce qu'on a additionné la sentinelle
          // undo l'opération
  moyenne /=  compteur;

  cout << "\nLe résultat est " << moyenne
       << endl << endl;

  return 0;
}
```

La boucle do-while est un cas spécial de la boucle while, lorsque le corps de la boucle doit être exécuté au moins une fois, mais que le nombre total de tours est indéterminé.

Un exemple est le déroulement d'un programme qui peut se répéter selon le choix de l'usager.

```
do
{
  tout_le_programme;
  cout << "\nVoulez-vous recommencer ? (O/N)" ;
  cin >> reponse;
}while( reponse == 'O' || reponse == 'o' );
```

Un autre exemple est la lecture d'un fichier qui requiert au moins une tentative de lecture avant de savoir que la fin du fichier est atteinte.

```
do
{
  fichier >> qqchose;
}while( !fichier.eof() );
```

La terminaison des boucles – condition, break et goto

Instruction break

L'instruction break termine la boucle immédiate qui la contient, par exemple, for(i=0; i<10;i++) if(toto) break; terminera la boucle for avant le temps, dès que toto sera vrai. Pour certains bouts de code, la solution du break est équivalente à la solution de la condition supplémentaire tout simplement parce que la condition doit être testée au moins une fois de toute façon.

```
while( 1 )
{
  partie_du_programme;
  cout << "\nVoulez-vous continuer ? (O/N)" ;
  cin >> choix;
  if(choix != 'o' && choix != 'O')
    break;
  suite_du_programme;
}
```

```
char reponse = 'o';
while( reponse == 'O' || reponse == 'o' )
{
  tout_le_programme;
  cout << "\nVoulez-vous recommencer ? (O/N)" ;
  cin >> reponse;
}
```

Dans d'autres exemples l'utilisation d'un break est plus avantageuse, par exemple lorsque la condition doit être testée deux fois, une fois dans le corps de la boucle pour le traitement des données, puis une autre fois pour voir si on a bien atteint la cible parmi toutes les données à analyser. C'est le cas de l'analyse du nombre premier, toutes les valeurs doivent être essayées avant de dire qu'un nombre est premier mais un seul diviseur trouvé permet de dire d'ores et déjà que le nombre N'est PAS premier. Il est possible à ce point de terminer la boucle directement avec le break, ou encore d'attendre le test de la condition et d'y tester à chaque itération le drapeau qui indique que le nombre semble encore être premier. Évidemment, un test à chaque itération est plus coûteux en temps d'exécution qu'un break exécuté une seule fois.

Vérifier si un nombre est premier

```
int premier = 1;  int nombre;  cin >> nombre;
for( int facteur=2; facteur<nombre/2; facteur++ )
{
  if(nombre%facteur = = 0)
  {
    premier = 0;
    break;  // arreter la boucle for
            // si le nombre a déjà un diviseur
  }
}
```

```
int premier = 1;  int nombre;  cin >> nombre;
for( int facteur=2; facteur<nombre/2 && premier; facteur++ )
{
  if(nombre%facteur = = 0)
    premier = 0;
}
```

Pour la recherche d'un élément, le principe est le même, aussitôt que l'élément est trouvé on peut terminer avec un break, ou encore attendre la condition évaluée à chaque itération de la boucle. La solution du break est plus économique.

Chercher un élément

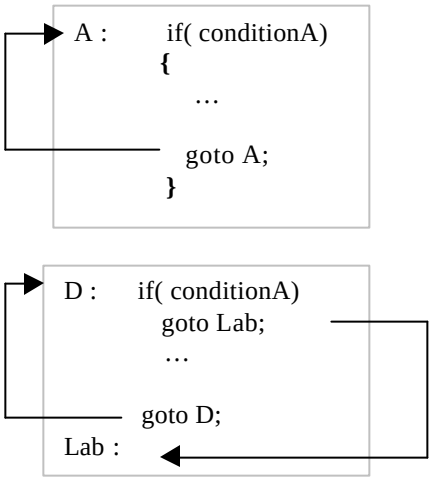
```
found = 0;
for( c=0; c<100; c++ ) // pour toutes les donnees
{
  if( data[c] == cle )
  {
    found = c;
    break;  // arreter le for si l'on trouve
  }
}
```

```
found = 0;
for( c=0; c<100 && !found; c++ )
{
  if( data[c] == cle )
    found = c;
}
```

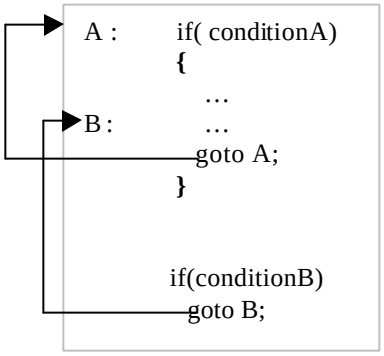
Le goto

Cette instruction diabolisée par la plupart des pédagogues en programmation, entraîne naturellement la formation du code que l'on appelle 'spaghetti', un code illisible avec des croisements dans les boucles. De tels croisement sont impossible en utilisant la syntaxe des boucles for, while et do-while.

Imitations de boucle while



Intersection entre 2 boucles (spaghetti)



Le goto, s'il est encore toléré dans le langage C, n'est pas qu'une structure maléfique, loin de là. Il permet d'éviter beaucoup de redondance dans les conditions d'arrêt de plusieurs boucles imbriquées et il est très comparable à l'instruction return qui s'utilise pour terminer une fonction un peu n'importe où dans le code.

En effet, si nous souhaitons cascader plusieurs boucles, disons trois, et les arrêter toutes les trois en même temps pour la même raison. Nous pourrions ajouter une condition à toutes les boucles mais il s'agit de tests redondants performés lors des itérations, donc de temps d'exécution perdu. De plus, dans ce cas particulier, l'utilisation du goto améliore la lisibilité du code.

```
for( x=0; x<80 && encore; x++)
  for( y=0; y<30 && encore; y++ )
    for( z=0; z<30 && encore; z++ )
      if(tab[x][y] ==  'O')
        encore = 0;
```

?

```
for( x=0; x<80; x++)
  for( y=0; y<30; y++ )
    for( z=0; z<30; z++ )
      if(tab[x][y] ==  'O')
        goto END;
END :
```

Si nous utilisions la technique du break dans le code ci-haut, cela terminerait la boucle for du z mais pas celle du y ni celle du x.

Analyse des boucles à l'aides de tableaux des valeurs

Pour débuter l'analyse d'une boucle, il importe de déterminer ses conditions initiales et la cause de son arrêt. Pour les boucles avec compteur, il est question de déterminer les bornes du compteur, c'est à dire sa **valeur initiale et la dernière valeur avec laquelle le corps de la boucle sera exécuté**.

Par exemple, **for(i = 0; i<10; i++)** exécutera son bloc avec i = 0, avec les valeurs intermédiaires ainsi qu'avec i = 9. Ensuite, i sera incrémenté à 10 mais à ce moment on n'entre plus dans le corps de la boucle, c'est la fin.

Un autre exemple où seulement l'opérateur relationnel est différent de l'exemple précédent, < remplacé par <=. Dans **for(i = 0; i<=10; i++)** les bornes du compteur changent, maintenant le corps de la boucle s'exécute avec i = 0 jusqu'à i = 10 **inclusivement**, cela fait un tour de plus. Bien sur, à la sortie de la boucle, i a la première valeur pour laquelle la condition était fausse, c'est à dire 11.

```
int i;
for(i = 0; i<=15; i++)
{
  cout << i << endl;

  switch(i)
  {
    case 3 : i += 5;  break;
    case 8 : cout << "Youpi" ; break;
    case 9 : cout << "Yoyo" << endl; break
    case 11 :  i++;
    case 14 :  i++;
  }
}

cout << "Derniere valeur de i est " << i;
```

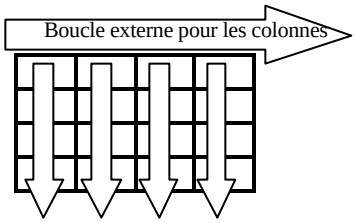
for	Affichage	switch	Affichage
0	**0**	-	
1	**1**	-	
2	**2**	-	
3	**3**	8	
9	**9**	9	**Yoyo**
10	**10**	-	
11	**11**	12	
13	**13**	-	
14	**14**	15	
16	**bloquée**		

```
0
1
2
3
9
Yoyo
10
11
13
14
Derniere valeur de i est 16
```

2 boucles imbriquées versus tableau

Les parcours de type carrés pour parcourir toutes les cases d'un tableau ou triangulaires pour parcourir les cases d'un triangle imaginaire dans un tableau 2D, ou encore pour faire des comparaisons d'éléments par éléments sont les types d'opérations nécessitant des boucles imbriquées.

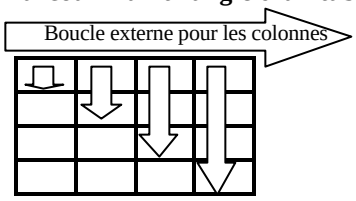
? **Parcourir un tableau 2D**



Boucle interne pour les rangees de chaque colonne

```
for(int col = 0; col< n_col; col++)
        for(int rang = 0; rang<n_rang; rang++)
                tableau[rang][col] ….
```

? **Parcourir un triangle d'un tableau 2D**



Boucle interne pour les rangees de chaque colonne

```
for(int col = 0; col< n_col; col++)
        for(int rang = col; rang<n_rang; rang++)
                tableau[rang][col] ….
```

? **Effectuer un travail pour chaque position d'une chaîne 1D nécessitant d'étudier chaque autre position (Combinatoire : Chaque personne doit donner une poignée de main à chaque autre position exactement une fois)**

- Tri avec comparaisons de chaque élément avec chaque autre élément une fois
- Recherche de paires identiques

Table 1 pointeur A

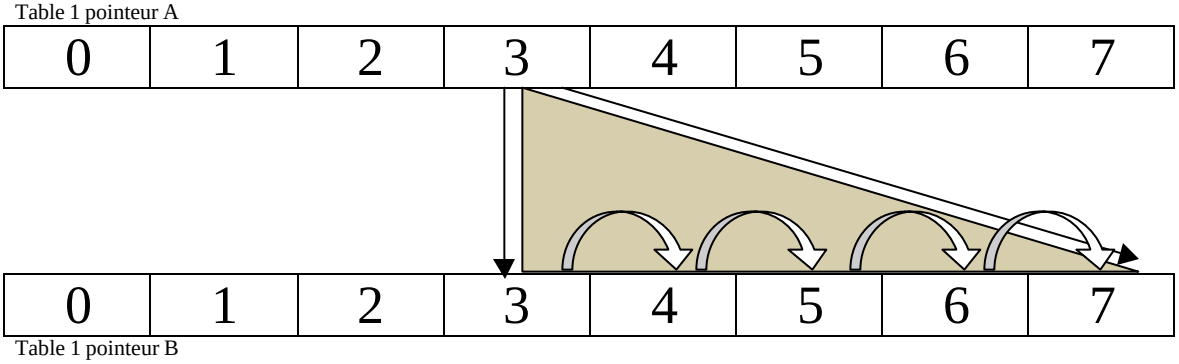


Table 1 pointeur B

Liste des comparaisons effectuées

A	B
0	1
0	2
0	3
0	4
0	5
0	6
0	7

A	B
1	2
1	3
1	4
1	5
1	6
1	7

A	B
2	3
2	4
2	5
2	6
2	7

Triangle illustré ci-haut

A	B
3	**4**
3	**5**
3	**6**
3	**7**

A	B
4	5
4	6
4	7

A	B
5	6
5	7

A	B
6	7

Exemple

```
int s=0;
for(int i=4; i>0; i--)
        for(int j=0; j<i; j++)
                s++;
```

i	j	s
4	0 ↓ 3 4 bloque (4 < 4)	1 2 3 4 4
3	0 ↓ 2 3 bloque (3 < 3)	5 6 7 7
2	0 1 2 bloque (2 < 2)	8 9 9
1	0 1 bloquée (1 < 1)	10 10
0 bloquée (0 > 0)	bloquée	10

Exercice

Écrire un code donnant l'affichage suivant :

```
4
3
2
1
2
3
4
```

MS-Dos ou Console
****
***
**
*
**
***
****

Une solution :

```
for(int ligne=4; ligne>=1; ligne--)
        for(int etoile=1; etoile<=ligne; etoile++)
                cout << '*';
for(int ligne=2; ligne<=41; ligne++)
        for(int etoile=1; etoile<=ligne; etoile++)
                cout << '*';
```

break et continue

break permet d'arrêter l'exécution de la boucle qui le contient instantanément. Si deux niveaux de boucles contiennent l'instruction, seule la boucle englobante, la plus interne sera arrêtée.

continue permet d'arrêter l'exécution de cette itération de la boucle qui la contient mais la rotation de la boucle continue selon les règles normales.

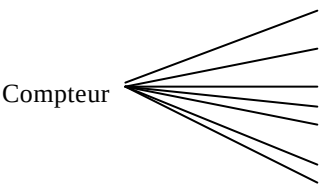
```
while(qqchose)
{
        if(donneeTrouvee( ))
                break;
}


for(int i = 0; i<5; i++)
{
        if(test)
                continue;  // ce continue bloque l'exécution des 2 lignes suivantes
                                // dans le cas ou test n'est pas 0
        tab[i] = 7;
        AjusterTest( );
}
```

En java, c'est encore plus complexe. Break et continue utilisés seuls ont les même signification. Cependant, il est possible de les utiliser avec des étiquettes et d'arrêter des boucles externes. Ceci permet de se sortir de la situation classique de l'utilisation du goto en C et C++, la sortie rapide de boucles imbriquées profondément.

```
BoucleExterne :
for(int i=0; i<10; i++)
{
        BoucleInterne :
        for(int j=0; j<10; j++)
        {
                if(i > j) continue BoucleExterne;
                if(i*j == 80) break Boucle Externe;
                // code
        }
}
```

Les boucles – Compteur et choix

Usages du compteur pendant la boucle, principalement positionnel



Position mémoire dure (fichier binaire)

Accès disque > accès mémoire

Position mémoire vive (tableau)
Position dans l'écran (tracer lignes, symboles, triangles de * ou pixels)
Position dans un intervalle (entre A et B dans N, nombres premiers, nombres parfaits)
Compte des répétitions d'un programme
Nombres d'éléments accumulés (moyenne, compte normal)
Contribution à la valeur dans les séries mathématiques

Usage à postériori du compteur

Compteur de données, lors d'entrées clavier ou lecture de fichiers, pour ensuite savoir la taille des structures de données.
Pour diviser les moyennes
Pour effectuer l'allocation dynamique

Structure syntaxique de la boucle while

```
while(condition pour continuer à exécuter les instructions)
{
        instructions à répéter
        + quelque chose qui rendra éventuellement la condition fausse (pour ne pas boucler infiniment)
}
```

Structure syntaxique de la boucle while

```
do
{
        instructions à répéter
        + quelque chose qui rendra éventuellement la condition fausse (pour ne pas boucler infiniment)
} while(condition pour continuer à exécuter les instructions);
```

Structure syntaxique de la boucle for

```
for(initialisation faite une fois; condition pour continuer ; incrémentations en instruction de clôture,
                                                                ce qui rendra la condition fausse)
{
        instructions à répéter
}
```

Quand utiliser un while au lieu de for ?

TANT QUE pas la fin du fichier
TANT QUE l'usager veut continuer
✍ événement indéterminé = utiliser la boucle while

Mais les deux reviennent au même logiquement et normalement en code machine(optimisation de boucle hardware), seule la présentation conceptuelle est différente,

Conditions de continuation de la boucle

Opérateurs de comparaison
== !=
< > <= >=

Combinaison de conditions
|| OU
&& ET

Comment écrire une boucle ?

1. Déterminer les opérations à répéter

Pour ce faire, il peut être utile, d'écrire temporairement sous forme séquentielle les premières opérations à effectuer avec la boucle. Par exemple, si je veux initialiser toutes les cases de mon tableau à zéro, je pense à :

```
Tab[0] = 0;  } ce qui se répète
Tab[1] = 0;
Tab[2] = 0;
…
```

Ici il n'y a qu'une ligne à répéter, et ceci avec une valeur variable pour l'indice du tableau.

2. Déterminer la valeur initiale et l'incrément

Déterminer ce qui est variable tout au long de la boucle, à partir du jet temporaire des premières lignes, déterminer la **valeur initiale** de la variable et son **incrément** avant la prochaine répétition des instructions

3. Choisir un type de boucle et écrire l'initialisation et l'incrément en utilisant sa syntaxe

BOUCLE FOR

Pour un nombre déterminé d'itérations, choisir préférablement une boucle for. Par exemple :

? Pour parcourir les cases d'un tableau de taille connue
? Pour répéter une opération dans un intervalle connu, comme entre 1 et un nombre à traiter

```
for(i = valeur_initiale;                    ; i += increment)
{
        Tab[ i ] = 0;
}
```

BOUCLE WHILE

Pour un nombre indéterminé d'itérations, choisir préférablement la boucle while. Par exemple :

? Pour lire un fichier de taille inconnu
? Pour faire répéter un programme aussi longtemps qu'on le désire
? Pour parcourir une structure de données dont la taille est inconnue, on parcourt jusqu'à la rencontre du signal de la fin des données, usuellement un pointeur null ou un caractère null.

```
i = valeur_initiale;
while(                    )
{
        Tab[ i ] = 0;
        i += increment;
}
```

4. Déterminer la condition de poursuite,
   l'exprimer avec un opérateur de comparaison (==, !=, <, >, <=, >=).

Il peut être utile d'écrire les deux dernière lignes attendues de la boucle, par exemple

```
Tab[ 98 ] = 0;
Tab[ 99 ] = 0;
```

À partir de cela, il faut penser à la condition qui doit continuer d'être vraie pour toutes les exécutions, mais qui doit devenir subitement fausse immédiatement après la dernière ligne désirée. Ici nous voulons la condition vraie pour i == 99 mais fausse pour la valeur suivante de i, c'est à dire pour i == 100

```
for(i = valeur_initiale;  i < 100  ; i += increment)
{
        Tab[ i ] = 0;
}

for(i = valeur_initiale;  i <= 99  ; i += increment)
{
        Tab[ i ] = 0;
}
```

Ces deux conditions répondent aux deux critères énoncés, soit
? Être vraie pour toutes les valeurs valides de i pour laquelle le corps doit s'exécuter
  Soit de 0 à 99.
? Être fausses pour la valeur suivante de i obtenue par l'incrémentation
  Soit 100

Même la condition **i != 100** serait valide sachant que la valeur de i au moment de la sortie de la boucle sera exactement 100, mais cette condition serait insécure advenant une erreur matérielle qui incrémenterait le i plus que prévu, la boucle deviendrait infinie.

5. Combiner plusieurs conditions de poursuite avec ET (&&) ou encore OU (||)

Seul les opérateurs de comparaisons, et la valeur des variable en soi, ont la possibilité de donner VRAI ou FAUX et d'être utilisé pour une condition d'exécution.

Pour la valeur des variable, elle s'évalue à vraie dès que cette valeur n'est pas zéro. Ainsi, la comparaison while(Variable != 0), n'est jamais nécessaire, et il est possible d'écrire seulement while(Variable).

```
while( Variable != 0 )     ?       while( Variable )
```

Maintenant que nous savons ce qui peut nous fournir les VRAI et les FAUX utiles, soit

- Les opérateurs ==, !=, <, >, <=, >=
- Les variables qui valent zéro ou non

Nous apprendrons à combiner ces conditions avec des && et des ||. Ces opérateurs sont binaires, c'est à dire qu'ils attendent deux arguments et ces arguments attendus sont les valeurs vraies ou fausses obtenues précédemment par les opérateurs de comparaisons. Par exemple :

```
( …      <=      …      &&      …      = =    … )

( …      >=      …      ||      …      !=     … )
```

5. Combiner plusieurs conditions de poursuite avec ET (&&) ou encore OU (||)

Utiliser le ET lorsque les deux conditions de poursuite doivent être vraies en même temps pour que l'action ait lieu. Par exemple, si une raison d'arrêter serait que l'on a parcouru toutes les données d'une structure, et que alternativement, le fait de trouver ce que l'on cherche devrait causer l'arrêt également, on peut dire que le fait de ne pas avoir encore trouvé (found = = 0) ET que le fait de ne pas encore avoir atteint le pointeur null (ptr != 0) sont nécessaires à la poursuite de la recherche. Ainsi la condition globale s'écrit (found = = 0 && ptr !=0).

&& - Pour **continuer**, les deux conditions doivent être **vraies en même temps**

Une autre façon de le formuler serait de dire que un seul des deux événements peut causer l'arrêt.

&& - **Un seul** des deux événements testés a le pouvoir de causer **l'arrêt**

Ainsi, lorsque une des deux conditions est absolument nécessaire à la poursuite des opérations, comme la présence de données à traiter, et que sa fausseté doit causer l'arrêt, comme par exemple la fin des données, le ET (&&) est utilisé.

? La fin d'une chaîne de caractères reconnue par son caractère null
while(str[i] != '\0' &&)

? La fin d'une liste chaînée, reconnue par un pointeur suivant qui est null.
while(ptr->suivant != 0 &&)

? La fin d'un fichier, reconnue par le caractère EOF
while((c = fgetc(nom)) != EOF &&)

? La fin d'un jeu, déterminée par les règles du jeu.
while(points > 0 && temps > 0 && choix == 'y')

? La présence dans un intervalle
if(x >= limite_inf && x <= limite_sup)

Utiliser le OU lorsque une seule des deux conditions de poursuite doit être vraie pour que l'exécution ait lieu. Par exemple, si une raison de continuer serait que l'usager a choisi 'y', et que alternativement, le fait de choisir 'Y' devrait permettre de poursuivre le programme de la même façon, on peut dire que le fait de d'avoir le choix 'y' (choix == 'y') OU que le fait d'avoir le choix 'Y' (choix == 'Y') est suffisant à la poursuite du travail. Ainsi la condition globale s'écrit
(choix == 'y' || choix == 'Y').

|| - Pour **continuer**, une seule des deux conditions doit être **vraie à la fois**

Une autre façon de le formuler serait de dire les deux événements sont nécessaire pour causer l'arrêt.

&& - **Les deux** événements testés peuvent ensemble causer **l'arrêt**

Ainsi, lorsque chaque condition est suffisante à la poursuite du programme, comme par exemple pour des réponses équivalentes, ou pour toute comparaison de la même variable avec l'opérateur == puisqu'une variable ne peut satisfaire à deux conditions différentes à la fois pour cet opérateur. En effet, choix ne saurait être égal à 'y' et à 'Y' en même temps.

? **Réponses ou valeurs équivalentes**
while(choix == 'y' || choix == 'Y')

? **Comparaisons de la même variable**
if(test = = 11 || test = = 14)

? **Extérieur d'un intervalle**
if(x < limite_inf || x > limite_sup)

Théorème de Morgan

On note que le dernier exemple avec OU est l'inverse du dernier exemple avec ET

Extérieur d'un intervalle
if(x < limite_inf || x > limite_sup) ? **if(!(x >= limite_inf && x <= limite_sup))**

Ou encore

La présence dans un intervalle
if(!(x < limite_inf || x > limite_sup)) ? **if(x >= limite_inf && x <= limite_sup)**

Ceci illustre bien le théorème de Morgan en logique booléenne qui stipule que une condition est équivalente à l'opposé (!) de son inverse (en interchangeant les ET et les OU et en interchangeant les opérateurs de comparaison pour leur inverse).

Opérateurs inverses

==	!=
<	>=
>	<=

Extérieur

(x > limite_sup)

X est plus grand que la limite sup
Condition suffisante

limite_sup

Intervalle

(x > limite_inf && x < limite_sup)

X est supérieur à la limite inf
Et inférieur à la limite sup

limite inf

Extérieur :

|| (x < limite_inf)

OU x est plus petit que la limite inf
Condition suffisante

Une partie de l'inverse de l'intérieur

! (x <= limite_sup)

X n'est pas plus petit que la limite sup
Condition nécessaire

limite_sup

Inverse de l'extérieur

! (x < limite_inf || x > limite_sup)

x ne fait pas partie de l'extérieur

limite inf

Une partie de l'inverse de l'intérieur

&& ! (x >= limite_inf)

ET x n'est pas plus grand que la limite inf. Condition nécessaire

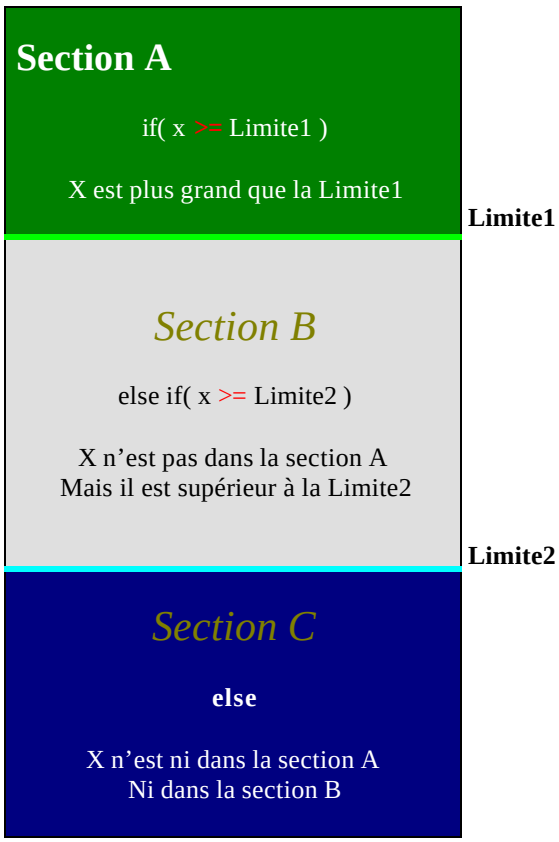
Minimiser le nombre de tests : Utiliser les conditions connues pour éviter des tests

On voit que dans la vérification d'intervalles, dans le else if il faut faire attention de ne pas retester ce qui a déjà été exclu par le fait qu'on est dans le else du premier if. Par exemple, si un if fait que les étudiants avec 90% et plus ont un 'A', le **else if** devrait assumer que c'est déjà moins de 90%. Par exemple, dans le bout de code suivant, la condition en relief est inutile.

if(note >= 90) { }
else if(**note < 90** et note >= 80)

Voici un autre exemple, pour vérifier que des paramètres biologiques sont bien à l'intérieur de limites données.

```
if( x >= Limite1 )
{

}
else if( x > Limite2 )
{


}
else
{

}
```




```
do
{

}while( nombre != 7 && nombre != 12 );

if( nombre = = 7 )
{

}
else // signifie que nombre est égal à 12
{

}
```

Enfin, si une condition d'arrêt contrôle une boucle, lorsqu'elle est arrêtée on sait pertinemment que cette condition est devenue fausse.

Si deux conditions d'arrêt contrôlent une boucle, lorsqu'elle est arrêtée, on sait que au moins une de ces deux conditions est fausse. Après en avoir testée une, si on voit qu'elle n'est pas la cause de l'arrêt, on peut déduire que c'est la seconde.

Exemple des nombres premiers

Définitions

Nombre premier 1 2 3 4 ……. N-1

Aucun diviseur exact

Un diviseur exact if(nombre % diviseur ==

Nombre pas premier 1 2 3 4 ……. N-1

Au moins un diviseur

```
// Entrer le nombre à vérifier
int n;
cout << "Entrez un nombre positif";
cin >> n;

// Déterminez si n est premier
int c = 2;
while( c<= n-1 )
{
        if( n % c == 0 )
                ici code pour indiquer que le nombre n'est pas premier et arrêter la boucle

}
```

Sauf 1

Sauf lui-même

Contrôle de la boucle : Tous les diviseurs sauf 1 et lui-même

Obtenir le nombre (n) à tester de l'usager (entrée au clavier)

Initialiser le drapeau à VRAI
(valeur par défaut car une seule trouvaille de facteur est suffisante pour signaler FAUX
mais tous les tests négatifs de facteurs sont nécessaires pour demeurer à VRAI)

POUR tous les diviseurs entre 2 et n-1 inclusivement

SI il y a un diviseur exact

Le nombre n'est pas premier, mémoriser l'événement dans le drapeau (FAUX)
Arrêter la boucle avec un break

SI le drapeau est vrai
Afficher que c'est un nombre premier

SINON
Afficher que ce n'est pas un nombre premier

```
int nombre;
int diviseur;
bool drapeau;

cout << "Entrez un nombre";
cin >> nombre;

drapeau = true;
diviseur = nombre –1;

while( diviseur >= 2 );
{
        if( nombre % diviseur == 0)
        {
                drapeau = false;
                break;
        }
        diviseur--;
}

if(drapeau)
        cout << nombre << " est premier";
else
        cout << nombre << " n'est pas premier";
```