Les instructions d'un programme sont exécutées dans l'ordre où elles sont écrites par le programmeur. C'est une exécution dite séquentielle. Ces instructions décrivent le traitement effectué par le programme.

Par contre, il est souvent nécessaire de modifier cet ordre d'exécution. Pour cela on dispose, comme dans tout langage de programmation, de structures permettant d'interrompre cet ordre séquentiel, en fonction de l'état de certaines variables.

Il existe deux types de structures :

- Les structures conditionnelles : sélections alternatives, sélections multiples (Instructions de test),
- Les structures répétitives : itératives (Instructions de boucle).

Les instructions de contrôle

Les instructions de contrôle définissent la suite des instructions à donner suivant le résultat d'une condition. Les instructions de contrôle peuvent être des expressions quelconques qui fournissent un résultat numérique de type scalaire. Toutefois, la plupart du temps, ces instructions de contrôle servent à évaluer et tester si une condition est vraie ou fausse, ce qui sous-entends que ces instructions seront plutôt des expressions de type booléen.

Type scalaire :
Les variables ou les entités de type scalaire, sont des variables dont on peut déterminer le nombre exact de valeurs possibles. Ce nombre est limité et connu. Par exemple, le type char procure uniquement 256 valeurs possibles. D'une façon générale, tous les types entiers correspondent à ce genre de critère (bool, char, short, int).
Par contre, les types réels, ne sont pas des types scalaires. En effet, entre la valeur 0.1 et 0.2, il existe normalement une infinité de valeur.

Exemples :

A == B	// Contrôle l'égalité. Attention : l'opérateur d'égalité est '==' et non pas l'opérateur d'affectation '='.
A>5 && A<10	// Teste si : 5<A<10
C faux. . .	// Il est possible de tester le contenu d'une variable. Si elle est booléenne, le résultat est vrai ou // Si elle est scalaire, toute valeur autre que 0 est alors considérée comme vrai (il existe une valeur). // Dans le cas contraire, si C est égale à 0, on considère qu'il n'y a pas de valeur, donc elle est // fausse.
A = 5	// Attention, ce n'est pas un test d'égalité. Cette expression est tolérée. Nous avons d'abord une . // affectation de la valeur 5 vers la variable A qui est ensuite évaluée. Comme, il existe une valeur, // A←5, la réponse est considérée comme vrai.

La sélection alternative

La première structure alternative connue est la sélection qui est représentée dans le langage C++ par le mot réservé **if** et éventuellement du mot réservé **else**.

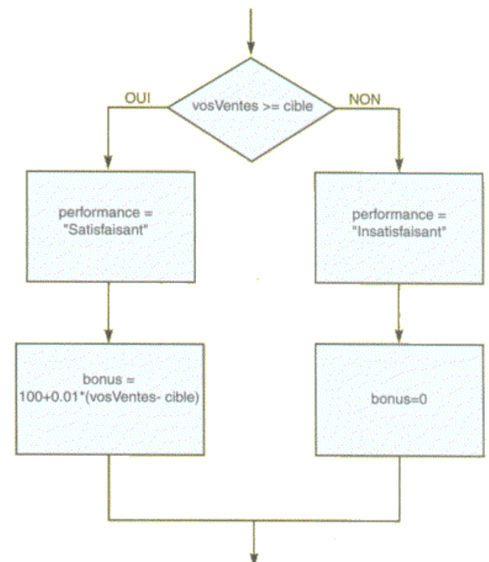
Syntaxe:	***Structure alternative en algorithme***	***Structure alternative en C++***
	Si *<instruction de contrôle>* ***alors*** *<bloc d'instructions 1>* ***Sinon*** *<bloc d'instructions 2>* ***FinSi***	***if*** *(<instruction de contrôle>) {* *<bloc d'instructions 1>* *}* ***else*** *{* *<bloc d'instructions 2>* *}*

La partie <bloc d'instructions> peut désigner :

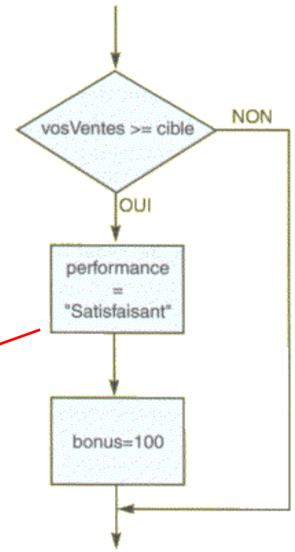
- Un (vrai) bloc d'instructions compris entre accolades.
- Une seule instruction terminée par un point-virgule.

Exercices :

Algorithme	***C++***
Si *vosVentes >= cible* ***alors*** *performance ← « Satisfaisant »* *bonus ← 100 + 0,01x(vosVentes-cible) ;* ***Sinon*** *performance ← « Insatisfaisant »* *bonus ← 0* ***FinSi***	***if*** *(vosVentes>=cible)* ***{*** *performance = « Satisfaisant » ;* *bonus = 100 + 0,01*(vosVentes-cible) ;* ***} else {*** *performance = « Insatisfaisant » ;* *bonus = 0 ;* ***}***



Au lieu d'avoir un bloc d'instruction, il est possible de n'avoir qu'une seule instruction à réaliser au sein de la sélection :




```
Algorithme                          C++

Si a > b alors                      if (a>b)
   max ← a                             max = a ;
Sinon                               else
   max ← b                             max = b ;
FinSi
```

La partie **else** est facultative :

```
Algorithme                               C++

Si vosVentes >= cible alors              if (vosVentes>=cible) {
performance ← « Satisfaisant »             performance = « Satisfaisant » ;
   bonus ← 100                             bonus = 100 ;
FinSi                                    }
```

Attention : Comme la partie else est optionnelle, les expressions contenant plusieurs structures if et if - else peuvent mener à des confusions. L'expression suivante peut être interprétée de deux façons :

```
Pour N=0, A=1 et B=2,

    if (N>0)                              if (N>0)
       if (A>B)                              if (A>B)
          MAX=A ;                               MAX=A ;
       else                               else
          MAX=B ;                            MAX=B ;

    Max reste inchangé                    Max obtiendrait la valeur B
```

Convention :
En C++ une partie else est toujours liée au dernier if qui ne possède pas de partie else. Dans notre exemple, C++ utiliserait donc la première interprétation. Sans cette règle supplémentaire, le résultat de cette expression serait imprévisible. L'indentation est là pour nous aider à mieux visualiser notre structure de programme. Par contre, le compilateur n'en tient absolument pas compte, il ne suit que les règles édictées. C'est le développeur qui décide de sa représentation, ce qui fait que l'indentation proposée sur la deuxième interprétation est fausse.

Solution :
Toutefois, pour éviter toute confusion, il est souvent recommandé d'utiliser les accolades { }. C'est d'ailleurs nécessaire dans le cas où nous désirions obtenir la deuxième interprétation.

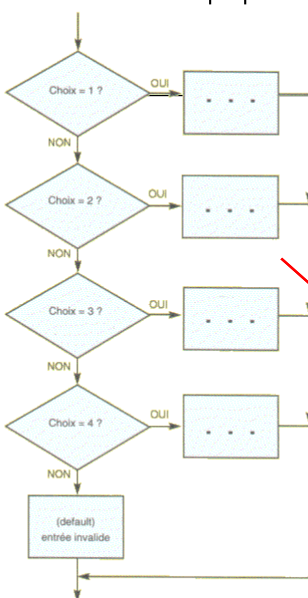
```
if (N>0)
{
  if (A>B)
     MAX=A ;
}
else  MAX=B ;
```

La forme → résultat = condition ? valeur à prendre si c'est vrai : valeur à prendre si c'est faux ;

Il est très fréquent de rencontrer l'écriture proposée à droite, où une seule variable change de valeur suivant le résultat d'un test. Il existe une syntaxe plus concise qui utilise le seul opérateur ternaire proposé par le langage C++, **'?:'**.
L'écriture est proposée à droite et il faut l'interpréter de la façon suivante : si A est plus grand que B, alors MAX prend la valeur de A, sinon MAX prend la valeur de B.

```
If (A>B) MAX=A ;
else  MAX=B ;
```

```
MAX = A>B ? A :
```



La sélection multiple

La sélection multiple permet de réaliser plusieurs cas de traitements possibles suivant la valeur d'une variable. Ce branchement conditionnel remplace avantageusement une suite d'instructions **if…else** sur une seule et même variable. L'opérateur qui traite la sélection multiple est le mot réservé **switch**. La sélection de la valeur est précédée du mot réservé case.

```
Algorithme                              C++

Suivant   Choix   faire                 switch (Choix) {
   1 : <bloc d'instructions 1>            case 1 : <bloc d'instructions 1> ; break ;
   2 : <bloc d'instructions 2>            case 2 : <bloc d'instructions 2> ; break ;
   3 : <bloc d'instructions 3>            case 3 : <bloc d'instructions 3> ; break ;
   4 : <bloc d'instructions 4>            case 4 : <bloc d'instructions 4> ; break ;
   autrement : <entrée invalide>          default : <entrée invalide > ; break ;
Fin Suivant                             }
```

Deux mots clés sont associés à l'utilisation du ***switch*** : ***break*** et ***default***. Le premier indique que l'on doit quitter le bloc. Le second permet de préciser le traitement à effectuer dans le cas où la variable n'est égale à aucune valeur répertoriée par les directives ***case***. L'instruction ***default*** est optionnelle.

> *ATTENTION : C++ exécute tout le code qui suit la directive case, Jusqu'à la sortie du switch ou jusqu'à une instruction break.*

Dans l'exemple ci-contre, notez qu'il n'y a une instruction break qu'à la fin de la première directive case. Dans ces conditions, les résultats de ce test sont les suivants :

```
int i = 0;
switch (a) {
   case 1   : i = i + 1; break;
   case 2   : i = i + 2;
   case 3   : i = i + 3;
   default  : i = i + 10;
}
```

- ***si a est égal à 1 :*** ***i = 1 ;***
- ***si a est égal à 2 :*** ***i = 15 ;***
- ***si a est égal à 3 :*** ***i = 13 ;***
- ***si a est différent de 1, 2 et 3 :*** ***i = 10.***

Ces résultats montrent qu'il est impératif d'utiliser le mot clé break si vous souhaitez uniquement exécuter le code associé à chaque directive case. Par ailleurs, ce switch permet de réaliser des opérations en cascade en jouant simplement sur l'ordre des tests et en omettant volontairement l'instruction break.

Les structures itératives

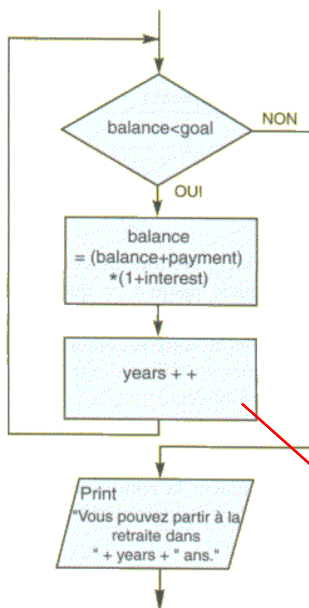
En algorithmique il existe trois structures de boucle principales, elles sont :

1. Tant que ...
2. Pour ...
3. Répéter ... Jusqu'à ...

En C++, ces structures sont quasi identiques à l'exception du Répéter ... Jusqu'à ..., qui en C++ se traduit plutôt par un Répéter ... Tant que ...

Chacune de ces itératives comporte une instruction de contrôle qui permet d'évaluer et de tester s'il est possible de continuer à traiter le contenu de l'itération. Il est alors impératif d'avoir en son sein, une variable relative à l'instruction de contrôle qui change d'état à chaque passage. Sans ce cas de figure, nous resterions indéfiniment au sein de l'itération.

L'itérative - Tant que …



Algorithme	***C++***
Tant que <instruction de contrôle> Faire	***while (<instruction de contrôle>) {***
<bloc d'instructions>	***<bloc d'instructions> ;***
Fin Faire	***}***

Je rappelle que l'instruction de contrôle désigne la condition qui, si elle est vraie, va nous permettre de rentrer dans la boucle. Si l'instruction de contrôle est fausse, on passe automatiquement à la suite du programme.

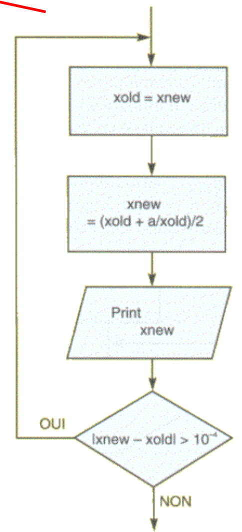
Une fois dans la boucle, on effectue le traitement tant que l'instruction de contrôle est vrai. Sitôt que cette condition n'est plus valide, à la fin de l'exécution du bloc d'instruction on sort de la boucle, et on effectue le traitement suivant.

Algorithme	***C++***
Tant que balance<goal Faire	***while (balance<goal) {***
balance ← (balance+payment)x(1+interest)	***balance=(balance+payment)*(1+interest);***
years ← years+1	***years++ ;***
Fin Faire	***}***
Ecrire 'Vous pouvez…'	***cout << « Vous pouvez… » ;***

L'itérative - Répéter… Jusqu'à

Algorithme	***C++***
Répéter	***do {***
<bloc d'instructions>	***<bloc d'instructions> ;***
Jusqu'à <instruction de contrôle>	***} while (complément de <instruction de contrôle>) ;***

Remarque :
Pour une structure `tant que ...`, suivant la condition de boucle, il peut arriver que l'on ne rentre jamais dans la boucle, et ainsi, le traitement interne n'est pas réalisé. Alors que pour une structure `Répéter ... Jusqu'à ...`, on est sûr d'effectuer au moins une fois le traitement interne, quelque soit la condition de sortie de la boucle.



Algorithme

```
Répéter
    xold ← xnew
    xnew ← (xold + a/xold) / 2
    Ecrire xnew
Jusqu'à (xnew-xold) > 10^-4
```

C++

```
do {
    xold = xnew;
    xnew = (xold + a/xold) / 2.0;
    cout << xnew;
} while ((xnew-xold) <= 0.0001 ) ;
```

Attention :
Je rappelle que l'itérative *Répéter…Jusqu'à* n'existe pas en langage C++. Elle se traduit par *Répéter… Tant que*. Du coup, il s'agit d'être très attentif quant à l'instruction de contrôle. En effet, le terme *Tant que* est la proposition inverse de *Jusqu'à*. Il est donc impératif de procéder de la même manière au sujet de l'instruction de contrôle. Dans l'exemple ci-dessus, vous remarquez que **>** a été remplacé par **<=** se qui correspond bien à la complémentation du contrôle.

Exercices :
Donnez le codage en C++ des structures proposées ci-dessous :

Algorithmes	*Résultats en C++*
Répéter Acquérir(valeur) Jusqu'à valeur = 2	
Répéter Acquérir(valeur) Jusqu'à valeur ≥ 12	
Répéter Acquérir(valeur) Jusqu'à valeur ≠ -23	
Répéter Acquérir(valeur) Jusqu'à -23 < valeur < 12	

L'itérative – Pour…

Cette structure permet d'effectuer un nombre fixe et déterminé d'itération.

Algorithme

```
Pour var de début à fin Faire
    < bloc d'instructions >
Fin Faire
```

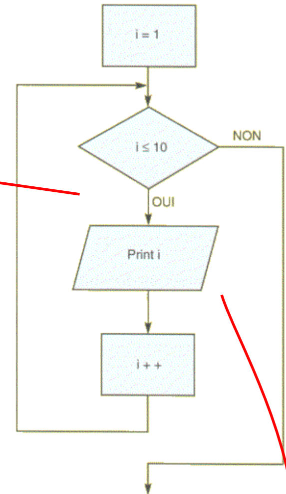
C++

```
for (initialisation ; test ; incrément ) {
    <bloc d'instructions> ;
}
```

Le début de la boucle for est constitué de trois éléments :

- ***initialisation***. C'est une expression qui initialise le début de la boucle. Quand le programme comprend un indice de boucle, cette expression peut le déclarer et l'initialiser, sous la forme « ***int i = 0*** ». Les variables déclarées dans cette partie de la boucle for sont locales par rapport à la boucle ; elles cessent d'exister une fois que la boucle a fini de s'exécuter. Vous pouvez initialiser plusieurs variables dans cette section en séparant chaque expression par une virgule. Ainsi, si l'instruction « ***int i = 0, j = 10*** » était utilisée dans cette section, elle déclarerait les variables i et j, et ces deux variables seraient locales par rapport à la boucle.
- ***test***. C'est le test qui s'exécute après chaque passage dans la boucle. Le test doit être normalement une expression booléenne telle que « ***i < 10*** ». Si le test a pour valeur ***true***, la boucle s'exécute. Dès que le test a pour valeur ***false***, la boucle cesse de s'exécuter.
- ***incrément***. C'est une expression. En général, l'incrément est utilisé pour modifier la valeur de l'indice de boucle afin de permettre à la boucle de s'approcher de plus en plus du stade qui lui permettra de retourner la valeur ***false***, et par conséquent de cesser de s'exécuter. Comme pour la section initialisation, il est possible de placer plusieurs expressions dans cette section en les séparant par une virgule.
- La partie ***traitement*** de la boucle for est l'instruction exécutée chaque fois que la boucle se répète. Comme pour les instructions if, vous pouvez inclure une instruction isolée ou un bloc.

Comme le montre l'organigramme ci-contre, le code suivant affiche les nombres 1 à 10 sur l'écran.



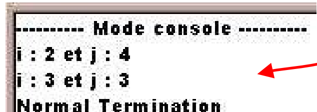
Algorithme	C++
Pour i de 1 à 10 Faire *Ecrire i* *Fin Faire*	*for (int i=1 ; i<=10 ; i++)* *cout << i ;*

Comme l'itérative ne contient qu'une seule instruction, il n'est pas nécessaire de créer un bloc d'instructions à l'aide des accolades { }. Il est également possible de créer des boucles ***décrémentales*** :

Algorithme	C++
Pour i de 10 à 1 Faire *Ecrire i* *Fin Faire*	*for (int i=10 ; i >0 ; i--)* *cout << i ;*

En fait, la partie incrément de la boucle for peut être de n'importe quelle nature. C'est une expression comme une autre. Ce qui veut dire que ce n'est pas obligatoirement une incrémentation ou une décrémentation. Comme je l'ai déjà évoqué dans la définition, il est également possible de placer des virgules, pour permettre une grande concision dans l'écriture.

```
for (int i=2, j=4 ; i<5 && j>2 ; i++, j-- )
    cout << "i = "<< i << " et j : "<< j;
```




```
int somme = 0;
for (int i=0 ; i<=100 ; i=i+1 )
    somme = somme + i;
```

```
for (int i=0, somme=0 ; i<=100 ; somme+=i );
```

En reprenant l'organigramme, vous avez sur la droite toute les possibiltés d'écriture quant à la boucle for, tout en donnant le même résultat. Toutefois, même si, effectivement, le C++ autorise et offre une grande souplesse d'écriture, il faut choisir la syntaxe qui paraît à la fois la plus lisible et qui offre en même temps une grande performance.

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

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

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

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

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

int i=1;
while ( i<=10 )  cout << i++;
```

Opérateurs de boucle

Pour terminer ce sujet, il existe deux opérateurs qui sont utilisés dans le cas des boucles : ***break*** et ***continue***.

On sort normalement d'une boucle dès que la condition n'est plus vérifiée. Il est cependant possible de prévoir une autre possibilité pour sortir d'une boucle, en utilisant l'instruction ***break***. Cette instruction a déjà été utilisée pour sortir d'un ***switch***, elle permet aussi de quitter une boucle.

L'instruction ***continue*** a pour objectif de remonter à la condition de la boucle. On peut ainsi ne pas exécuter une partie des instructions du corps de la boucle. Dans le cas d'une boucle ***for***, la troisième expression, qui correspond normalement à l'incrémentation, sera exécutée avant le retour au test. L'exemple suivant montre l'utilisation des deux mots réservés ***break*** et ***continue***.

```
1  while (true) {
2    switch (a) {
3      case 1 : /* instructions */  break;
4        // on sort du switch et pas de la boucle
5      case 2 : continue;
6        // remonte à la condition de la boucle
7    }
8    if (quitter == 1) break;
9      // on sort de la boucle
10 }
```

Le code ci-dessus illustre plusieurs notions liées à la manipulation des boucles, des tests et des deux mots clés ***break*** et ***continue***. Remarquez, sur la ligne ***1***, la condition de la boucle qui est égale à ***true***. Dans ce cas, la condition est toujours vraie, ce qui permet de définir ce que l'on nomme une boucle sans fin. Cette technique constitue certainement le moyen le plus simple pour mettre en œuvre une boucle, mais il est nécessaire dans ce cas de penser à mettre en place une condition de sortie. Le seul moyen de quitter cette boucle est donc d'utiliser l'instruction ***break***. On peut remarquer dans ce programme que ***break*** est employé à deux reprises (ligne ***3*** et ligne ***8***). Le premier ***break*** sert à sortir du ***switch***, mais pas de la boucle. Le second permet de quitter la boucle. La ligne ***5*** correspond à une instruction ***continue*** qui permet de remonter à la condition de la boucle (ligne ***1***) et n'a qu'un seul objectif, ne pas exécuter le code qui suit dans la boucle (situé après la ligne ***5***).