Tests et boucles

Tests

Comme dans tous les langages, il existe, en C++, un moyen pour effectuer une action en fonction de la valeur d'une variable ou d'un événement.

1) if

La plus simple ce présente comme suit :

```
if(condition)
{
  action
}
```

Si on devait trouver un équivalent à **if** en français, on dirait 'si'. **Si** condition est vrai, alors action est exécutée.

Imaginons que l'on dispose d'une variable **int** nommée **age**.

```
if(age > 18)
{
  cout << "Vous êtes majeur!!!";
}
```

Si la valeur de la variable **age** est plus grande que 18, 'Vous êtes majeur !!!' est affiché. Les opérateurs de comparaison disponibles sont :

Opérateur	Fonction	Exemple vrai	Exemple faux
<	Strictement plus petit	12 < 14	12 < 12
<=	Plus petit ou égal	12 <= 14	12 <= 6
>	Strictement plus grand	14 > 12	14 > 14
>=	Plus grand ou égal	12 >= 12	12 >= 14
== (double =)	Strictement égal	12 == 12	12 == 6
!=	Différent	12 != 2	12 != 12

Si l'expression est vraie, on dit qu'elle vaut **true** si elle est fausse, on dit qu'elle vaut **false**.

Il faut remarquer qu'il n'y a pas de ; après l'instruction **if**.

2) else

Il est possible de mettre un **else** après le **if**. Le **else** exprime 'sinon' : Si la condition du **if** n'est pas satisfaite, c'est l'instruction du **else** qui est exécutée. L'instruction **else** n'a pas de condition.

Voici le cas général :

```
if(condition)
{
  action ;
}
else
{
  action_par_defaut ;
}
```

Voici un exemple concret :

```
if(age == 18) // vous remarquerez les double =
{
  cout << "Vous avez 18 ans!!!" ;
}
else
{
  cout << "Vous n'avez pas 18 ans" ;
}
```

On a donc un test spécifique et dans tous les autres cas, **else** est la.

3) else if

Il existe un troisième mot clé, c'est **else if** il permet d'avoir des alternatives au **if** de départ.

Voici sa syntaxe générale :

```
if(condition_1)
{
  action_1
}
else if(condition_2)
{
  action_2
}
else if(condition_3)
{
  action_3
}
else
{
  action_par_defaut
}
```

On peux avoir autant de **else if** que l'on veux mais il doit exister au moins un **if** pour un (ou des) **else if** qui doivent se suivre.

Reprenons notre variable age :

```
if(age < 3)

  cout << "Vous etes bebe " ;
}
else if(age == 4 )
{
  cout << "Vous avez 4 ans" ;
}
else if(age < 13)
{
  cout << "Vous etes jeune" ;
}
else if(age < 17)
{
  cout << "Vous etes adolescent" ;
}
else
{
  cout << "Vous etes adulte" ;
  cout << "Ceci etait l'action par defaut" ;
}
```

Ce qui est très important c'est qu'une fois qu'une condition a été validée, les autres conditions ne sont pas testées.
Dans notre exemple, si **age** vaut 4, le programme ne fera pas les tests qui se trouvent après, il passera à la commande située après la dernière condition. Dans notre cas c'est le **else**. Cette manière de procéder apporte un gain de temps au moment de l'exécution.

Il est possible de placer autant d'actions que l'on veux entre les crochets. De plus, si un test ne contient qu'une action, il est permit de ne pas mettre les accolades.

Nous aurions donc pût écrire :

```
if(age < 3)
  cout << "Vous etes bebe " ;
else if(age == 4 )
  cout << "Vous avez 4 ans" ;
else if(age < 13)
  cout << "Vous etes jeune" ;
else if(age < 17)
  cout << "Vous etes adolescent" ;
else
{
  cout << "Vous etes adulte" ;
  cout << "Ceci etait l'action par defaut" ;
}
```

Seul le block **else** a des accolades car il contient plus d'une action. Grâce à la suppression des accolades, nous avons un gain de place considérable et un code moins lourd.

4) and et or

Il est possible de combiner des conditions pour avoir des tests plus précis. Les deux opérateurs sont **&&** et ||. L'opérateur **&&** rempli le rôle de **et** (**AND**) tandis que l'opérateur || rempli le rôle de **ou** (**OR**).

Prenons deux conditions :

condition_1	condition_2	condition_1 && condition_2	condition_1 || condition_2
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

Pour que **&&** soit true, il faut que les deux conditions soient acceptées tandis qu'il n'en faut qu'une avec ||.

Voyons un exemple de code, toujours avec notre variable **age** :

```
if(age >= 0 && age <= 3) // si age est plus grand ou égal à 0 ET que ag est plus petit
                         // ou égal à trois
{
  cout << "Vous avez entre 0 et 3 ans";
}

if(age == 17 || age == 18) // si age est égal à 17 OU si age est égal à 18
{
  cout << "Vous avez 17 ou 18 ans";
}
```

5) un raccourci

Dans le cas d'un block **if-else** simple, il est possible d'utiliser un raccourci.

Prenons :

```
int a = 100, b ;

if(a < 1000)
  b = 1 ;
else
  b = 0;
```

Il est possible d'écrire le code ci-dessus sous la forme :

```
int a = 100, b ;

b = a < 1000 ? 1 : 0;
```

C'est la même chose, si **a** est plus petite que 1000, **b** vaut 1. Sinon, **b** vaut 0.

6) switch

Imaginons maintenant que nous ayons une action pour l'age 0, une pour l'age 1, une pour l'age 2 et ainsi de suite jusqu'à l'age 10. Un code avec des **if** serait très lourd. Il existe un moyen de faire plus léger : une structure **switch**.

Avec une structure **switch** il n'est possible de tester que l'égalité.

Selon la norme du C, un **switch** peux contenir 257 conditions, C++ recommande d'en prendre en charge au moins 16384.

Voici la syntaxe générale :

```
switch(variable_a_tester)
{
  case valeur_1:
  action_1;
  break;

  case valeur_2:
  action_2;
  break;

  default
  action_par_defaut;
}
```

Dans le **switch**, des actions sont prévues pour les valeurs désirées. On commence par le mot clé **case**, on ajoute la valeur que doit être la valeur testée, et enfin, on met **:**. Après la liste des actions, il ne faut pas oublier le mot clé **break** qui sort du **switch** une fois une action exécutée.

L'étiquette **default** est équivalent au **else**, c'est le cas par défaut.

Voici un exemple concret, encore et toujours avec notre variable **age** :

```
switch(age)
{
  case 1:
  cout << "Vous avez 1 an" ;
  break ;

  case 2 :
  cout << "Vous avez 2 an" ;
  break ;

  case 3 :
  cout << "Vous avez 3 ans" ;
  break ;

  default :
  cout << "Vous avez plus de 3 ans" ;
}
```

Les tests sont utilisés tout le temps en programmation pour faire un programme qui réagisse en fonction de données et d'actions.

Dans nos exemples, nous avons testé des variables, mais il est possible de tester l'état de contrôles, par exemple, dans un programme Win32, savoir si un bouton est coché ou non ou encore savoir si l'ordinateur est connecté à internet… Les possibilités sont infinies.

Les boucles

Nous allons passer maintenant aux boucles, elles servent à répéter une action un certain nombre de fois. Il en existe plusieurs.

1) for

Sa syntaxe est la suivante :

```
for(variable_et_initialisation; condition; incrementation)
{
  Action
}
```

Une boucle **for** contient donc 3 champs. Aucun n'est nécessaire mais les **;** qui les séparent le sont.

- Le premier champ correspond a l'initialisation de la variable de boucle, elle peux être déclarée à ce moment la.
- Le deuxième champ est une conditions, c'est tant que cette conditions est vraie que la boucle est exécutée.
- Le troisième champ est destiné à l'incrémentation de la variable de boucle.

Voici un exemple qui affiche les nombres de 1 à 100 dans la console, 5 par ligne.

```
// on déclare une variable i, initialisée à 1
// on exécutera la boucle tant que i est inférieure ou égale à 100
// on incrémente i à chaque passage de boucle
for(int i = 1; i <= 100; i++)
{
  // on affiche i
  cout << i;

  // si le reste de la division de i par 5 vaut 0, on passe a la ligne
  if(i % 5 == 0)
    cout << "\n" ;
}
```

La variable de boucle est souvent notée **i** ou **j**, par convention.

Si aucun des champs n'est rempli, il s'agit d'une boucle infinie…

Une boucle infinie n'a pas de grand intérêt, à part de bloquer le programme. Nous verrons qu'il existe des instructions pour sortir d'une boucle.

```
// cette boucle se répétera indéfiniment
for(;;)
{
  cout << "…\n" ;
}
```

Voici l'exemple d'un boucle **for** où le premier et le troisième champs ne sont pas remplis.

```
// on déclare et on initialise la variable avant la boucle
int i = 1 ;

for( ; i <= 100 ; )
{
  // on affiche i
  cout << i;

  // si le reste de la division de i par 5 vaut 0, on passe a la ligne
  if(i % 5 == 0)
    cout << "\n" ;

  // on incrémente la variable
  i++ ;
}
```

2) while

La deuxième boucle que nous allons voir est la boucle **while**.

```
while(condition)
{
  action
}
```

Tant que **condition** est vraie, **action** est exécutée. **Condition** peux contenir plusieurs sous-conditions liées par **&&** et ||. L'initialisation de la vraible de boucle doit se faire avant la boucle.

Reprenons notre code qui affiche les nombres de 1 à 100 ;

```
int i = 1;

// tant que i est plus petit ou égal à 100
while(i <= 100)
{
  // on affiche i
  cout << i;

  // si le reste de la division de i par 5 vaut 0, on passe a la ligne
  if(i % 5 == 0)
    cout << "\n" ;

  // on oublie pas l'incrémentation sinon c'est la boucle infinie
  i++;
}
```

Par convention, la syntaxe pour une boucle **while** infinie est la suivante :

```
while(1)
{
  action;
}
```

3) do-while

Regardons le code précédant. Si la variable **i** est supérieure à 100 à l'entrée de la boucle, elle n'est pas exécutée Il se peux que le programmeur veuille que sa boucle soit exécutée **au moins une fois**. C'est le rôle de la boucle **do-while**.

Le boucle **do-while** se comporte comme la boucle **while**.

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

Voici un exemple concret :

```
int i = 1;

do
{
  // on affiche i
  cout << i;

  // si le reste de la division de i par 5 vaut 0, on passe a la ligne
  if(i % 5 == 0)
    cout << «\n» ;

  // on oublie pas l'incrémentation sinon c'est la boucle infinie
  i++;
}while(i <= 100); // il ne faut pas oublier le ;
```

4) Les mots clés des boucles

Il existe deux mots clés qui modifient le déroulement normal des boucles, il s'agit de **break** et de **continue**. Ces deux mots clés sont applicables aux trois boucles que nous avons vues.

break permet de sortir d'une boucle, soit pour sortir prématurément d'une boucle en cas d'erreur ou lorsqu'on est arrivé à un résultat souhaité, soit pour quitter une boucle infinie…

```
while(1)
{
  // on sort de la boucle
  break;

  // cette instruction ne sera jamais exécutée
  cout << "Hello world!!!";
}
```

continue permet de passer a la prochaine itération de la boucle, en sautant toutes les instructions qu'il y a jusqu'à la fin de la boucle.

```
for(int i = 0; i <= 100; i++)
{
  cout << "a ";

  continue;

  // b ne sera jamais affiché
  cout << "b ";
}
```

Dans ce cas, **a** sera affiché 101 fois, mais **b** ne sera jamais affiché. C'est comme si on avait un **goto debut_de_boucle** a la place de **continue**.

5) Informations supplémentaires

Les boucles sont très pratiques, mais il faut y faire attention. Comme un groupe d'instructions est exécuté un certain nombre de fois, s'il n'est pas optimisé, beaucoup de temps est perdu.

Si une opération inutile est exécutée à chaque passage de boucle, il faut essayer de l'extraire.

Exemple :

```
for(int i = 0; i <= 100; i++)
{
  int x = 2 * 4 * 6 * 10 * i;

  cout << x;
}
```

Dans cet exemple, à chaque itération **2 * 4 * 6 * 10** est calculé alors que c'a n'est pas nécessaire, il faut donc extraire ce calcul :

```
int calcul = 2 * 4 * 6 * 10;

for(int i = 0; i <= 100; i++)
{
    int x = calcul * i;

    cout << x;
}
```

Il est possible d'optimiser le test de la boucle dans certains cas. Si on a :

```
int j = 0;

while(j < 100)
{
  cout << "x";

  j++;
}
```

Ce programme affiche 100 fois un **x**.

On pourrait remplacer le code par :

```
int j = 100;

while(j != 0)
{
  cout << "x";

  j--;
}
```

On peux se demander quel intérêt de choisir une version plutôt que l'autre.
Regardons comment se passe le **j < 100** du premier code :

1) 100 est soustrait à j
2) On compare la valeur obtenue avec 0, si c'est plus grand, **j** est plus petit que 100

On a donc deux opérations. Dans le deuxième code, on a juste la comparaison avec 0, qui est un test spécial sur les processeurs, on économise donc la soustraction. ☺ ☺ ☺

6) Un programme pour résumer tout ça

J'imagine que tout ce cours, sans exemple concret, c'est pas utile du tout, voici donc un petit programme qui va résumer le tout.

Il s'agit du nombre mystérieux. On prend un nombre, et l'utilisateur doit le trouver. On lui donne des indices selon ce qu'il entre. Finalement, on affiche le nombre d'essais nécessaires pour parvenir au résultat.

Il sera malheureusement nécessaire de rentrer le nombre à chercher dans le code, ce qui n'est pas très pratique…

```
#include <iostream>
#include <conio.h>

using namespace std;

int main()
{
  // ------------------------ //
  // DECLARATION DES VARIABLES //
  // ------------------------ //
  // nb_coups: nombre de coups nécessaire pour arriver au resultat
  // nb_cache: le nombre à trouver
  // nb_entre: le nombre qui est entre par l'utilisateur
  int nb_coups = 0, nb_cache = 63, nb_entre;

  // message de bienvenue
  cout << "-----------------------------------------\n";
  cout << "Bienvenue dans le programme du nombre cache\n";
  cout << "-----------------------------------------\n\n";

  // information
  cout << "Le nombre est compris entre 0 et 100\n\n";

  // boucle infinie
  while(1)
  {
    cout << "Entrez un nombre: ";
    cin >> nb_entre;

    // si le nombre entré est en dehors des nombres autorisés
    if(nb_entre < 0 || nb_entre > 100)
    {
      cout << "Le nombre doit etre compris entre 0 et 100\n\n";

      // on passe a la prochaine itération, ca nous évite
      // de comptabiliser un coup
      continue;
    }
```

```
    // ca fait un coup en plus
    nb_coups++;

    // si on entre un nombre trop petit
    if(nb_entre < nb_cache)
      cout << "Le nombre a trouver est plus grand que " << nb_entre << "\n\n";
    // si on entre un nombre trop grand
    else if(nb_entre > nb_cache)
      cout << "Le nombre a trouver est plus petit que " << nb_entre << "\n\n";
    // alors le bon nombre a été entré
    else
      // on sort de la boucle
      break;
  }

  cout << "Felicitions!!! Vous avez trouve le nombre cache en " ;
  cout << nb_coups << " essai(s)\n\n";

  cout << "Appuyez sur une touche pour terminer le programe...";
  getch();

  return 0;
}
```

Ouf, voici enfin la fin de ce long chapitre… Bon coding…