



Centre Informatique pour les **L**ettres
et les **S**ciences **H**umaines

Apprendre C++ avec Qt : Leçon 4

Structures de contrôle

1 - Expressions et opérateurs booléens.....	2
Opérateurs de comparaison.....	2
Opérateurs logiques	2
Expressions booléennes complexes.....	3
2 - Conditions	4
Exécution conditionnelle simple	4
Exécution alternative.....	4
Expression alternative	5
Exécution à choix multiple	5
3 - Itérations	6
La boucle while()	6
La boucle do ... while();	7
La boucle for(; ;)	7
Conversion des structures itératives les unes en les autres	8
4 - Bon, c'est gentil tout ça, mais ça fait déjà 7 pages. Qu'est-ce que je dois vraiment en retenir ?	8

Maintenant que nous savons définir des variables et des fonctions, il serait peut-être temps d'enrichir un peu notre répertoire d'instructions, de façon à pouvoir écrire des programmes effectuant des traitements un peu moins élémentaires !

Un des outils qui nous fait le plus défaut (au point, d'ailleurs, que le TD 03 ait exigé que nous anticipions sur le cours) est la possibilité d'influer sur le déroulement de l'exécution du programme. La règle générale d'**exécution séquentielle** (les instructions sont exécutées les unes après les autres, dans l'ordre dans lequel elles apparaissent dans le programme) peut être remise en cause par deux types de structures de contrôle : les unes permettent de décider si un bloc d'instructions doit ou non être exécuté (c'est ce qu'on appelle l'**exécution conditionnelle**), alors que les autres permettent de spécifier qu'un bloc d'instructions doit être exécuté plusieurs fois de suite (c'est ce qu'on appelle l'**exécution itérative** ou, plus simplement, une boucle). Ces deux types de structure de contrôle nécessitent l'évaluation d'expressions booléennes ou, en d'autres termes, exigent qu'on exprime des conditions dont le processeur pourra déterminer si elles sont ou non remplies au moment de l'exécution de la ligne de code qui les contient. Il nous faut donc commencer par apprendre à exprimer de telles conditions.

1 - Expressions et opérateurs booléens

La façon la plus simple d'exprimer une condition est sans doute une expression procédant à la comparaison directe de deux termes, mais il est également possible d'effectuer des opérations logiques sur des valeurs booléennes.

Opérateurs de comparaison

Le langage C++ propose six opérateurs dyadiques¹ permettant d'effectuer différentes comparaisons directes. Leur syntaxe est identique :

expressionUn **opérateur** **expressionDeux**

Une expression ainsi formée a pour valeur **true** si la relation spécifiée par l'opérateur est vérifiée, **false** dans le cas contraire. Les opérateurs disponibles sont donnés par le tableau ci-contre.

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

```
//exemples de calculs booléens utilisant les opérateurs de comparaison
1 bool v1, v2; //a et b sont des variables numériques supposées précédemment définies
2 v1 = (a < b); //true si le contenu de a est plus petit que celui de b
3 v2 = (a != b); //true si les deux variables ont des valeurs différentes
```

Attention à ne pas confondre l'opérateur de test d'égalité avec l'opérateur d'affectation : comme les expressions bâties avec l'opérateur d'affectation ont une valeur (cf. Leçon 3) et que les valeurs numériques peuvent être converties automatiquement en valeurs booléennes (cf. Leçon 2), le compilateur n'émettra aucune objection s'il rencontre une séquence telle que :

```
//a et b sont des variables numériques supposées précédemment définies
bool v = (a = b); //ne serait-ce pas plutôt : bool v = (a == b) ?
```

Dans cet exemple, la variable v prend la valeur false si b contient 0, et true sinon. Il n'est pas certain que ce soit là ce que l'auteur du programme avait en tête...

Opérateurs logiques

Tout comme les opérateurs arithmétiques conduisent à des résultats numériques par combinaison de données numériques, les opérateurs logiques conduisent à des résultats booléens par combinaison de données booléennes. C++ propose deux opérateurs logiques dyadiques (ET et OU) et un opérateur logique monadique (NON).

Le **OU inclusif**, noté **||**, produit des expressions qui sont vraies dès lors que l'un des opérandes est vrai (ou, autrement dit, qui ne sont fausses que si les opérandes sont tous deux faux).

¹ Un opérateur est dit dyadique lorsqu'il suppose la présence de deux opérandes pour former une expression valide. Il arrive qu'un tel opérateur soit qualifié de *binnaire*, mais ce terme présente une certaine ambiguïté (cf. Leçon 24). Un opérateur ne nécessitant qu'un opérande pour former une expression valide est dit monadique (ou *unaire*), alors qu'un opérateur exigeant trois opérandes sera dit *ternaire* (et non triadique, sans doute par crainte des mafias chinoises :).

```

1 //exemple de calcul booléen utilisant le OU logique inclusif
2 bool v1; //(a et b sont des variables numériques supposées précédemment définies)
v1 = (a > b) || (a == b); //équivalent à v1 = (a >= b);

```

Le caractère | a pour code ASCII 124. Si vous avez du mal à le trouver sur votre clavier, souvenez-vous que Windows permet de saisir n'importe quel caractère en tapant son code ASCII sur le pavé numérique pendant que la touche Alt est maintenue enfoncée. Il est aussi possible (cf. [Annexe 1](#)) d'utiliser le mot or (ou, en anglais) en lieu et place du symbole ||.

Il n'existe pas, en C++, d'opérateur booléen correspondant au **OU exclusif**. Si vous avez besoin d'une expression qui soit vraie lorsqu'une et une seule de deux expressions logiques est vraie, utilisez simplement (expression1) != (expression2). En effet, comme il n'existe que deux valeurs booléennes, la seule situation où les deux expressions ont des valeurs différentes est celle où l'une est vraie alors que l'autre est fausse.

Le **ET logique**, noté &&, produit des expressions qui ne sont vraies que lorsque les opérandes sont tous deux vrais (ou, autrement dit, qui sont fausses dès que l'un des opérandes est faux).

```

1 //exemple de calcul booléen utilisant le ET logique
2 bool v1; //(a et b sont des variables numériques supposées précédemment définies)
v1 = (a > 3) && (b < a);

```

Dans une expression formée avec && ou ||, le second opérande n'est évalué que lorsque la valeur du premier ne permet pas, à elle seule, de déterminer la valeur de l'expression. Ainsi, dans le cas de

```
(a != 0 && (b/a) > c)
```

si a vaut 0, l'expression est fausse de toutes façons et la sous-expression (b/a) n'est pas évaluée, ce qui évite l'erreur d'exécution que causerait la tentative de division par 0.

La **négation logique**, notée !, produit des expressions qui sont vraies lorsque l'opérande est faux (ou, autrement dit, qui sont fausses lorsque l'opérande est vrai).

```

1 //exemple de calcul booléen utilisant la négation
2 bool v1; //(a et b sont des variables numériques supposées précédemment définies)
v1 = !(a > b); //équivalent à v1 = (a <= b);

```

Expressions booléennes complexes

Etant donné que la combinaison de deux expressions booléennes à l'aide d'un opérateur logique est elle-même une expression booléenne, il est possible d'enchaîner plusieurs opérations logiques, comme par exemple dans l'expression suivante :

```
(a == 3 || a == 27 || a == 42) //vraie si a contient l'une des 3 valeurs
```

Il est toutefois conseillé de limiter strictement ce genre d'expressions aux cas où un même opérateur logique intervient plusieurs fois. En effet, lorsqu'une expression fait intervenir à la fois des OU et des ET (sans même parler de la négation), il se pose un problème d'ordre d'évaluation et l'expérience montre que le risque d'erreur humaine (lors de l'écriture ou de la relecture du programme) croît d'une façon tout à fait disproportionnée en regard des avantages que l'on avait initialement cru voir dans cette façon de s'exprimer.

Même lorsqu'un seul opérateur logique est impliqué, l'usage d'expressions complexes pose souvent des problèmes aux débutants. Une erreur classique est d'oublier que les opérateurs de comparaisons sont dyadiques et d'essayer d'exprimer la condition précédente en écrivant :

```
(a == 3 || 27 || 42) //vraie quelle que soit la valeur de a !
```

Comme les valeurs numériques sont, en cas de besoin, automatiquement converties en valeurs booléennes (cf. [Leçon 2](#)), l'expression ci-dessus est syntaxiquement correcte et le compilateur ne signalera aucun problème. (Puisque 27 est non nul, il sera converti en true et l'expression, où il figure comme opérande d'un OU, sera donc vraie quelle que soit la valeur des autres opérandes.)

Lorsque la situation exige des calculs logiques un peu complexes, il est souvent préférable de stocker, dans des variables judicieusement baptisées, des résultats intermédiaires obtenus à l'aide d'expressions simples : l'écriture, la mise au point et la relecture du programme s'en trouvent largement facilitées.

2 - Conditions

Les expressions dont la valeur est un booléen se prêtent naturellement au contrôle de l'exécution de blocs d'instructions.

Exécution conditionnelle simple

La façon la plus directe d'appliquer ce principe est peut être de n'exécuter un bloc que si une expression est vraie. Cet effet est obtenu très simplement, à l'aide du mot `if` précédant l'expression booléenne placée entre parenthèses :

```
if (expression)
    bloc
```

Lorsque le bloc ne comporte qu'une instruction, celle-ci peut ne pas être incluse entre accolades, ce qui allège un peu l'écriture mais peut, dans certains cas, être une source d'erreurs de programmation².

```
//exemple d'exécution conditionnelle simple
//(a et b sont des variables numériques supposées précédemment définies)
1 if(a > b)      //remarquez l'absence de point virgule sur cette ligne
2 {
3     a = b;     //ces instructions ne seront exécutées que si le contenu de a
4     b = 4;     //est plus grand que celui de b
5 }
6 b = 2 * b ;    //cette instruction est exécutée dans tous les cas
```

Il n'y a, bien entendu, aucune restriction particulière sur le contenu du bloc dont l'exécution est conditionnelle (il peut, notamment, comporter lui-même un sous-bloc dont l'exécution dépend, elle aussi, d'une condition).

Exécution alternative

Un perfectionnement possible de l'idée d'exécution conditionnelle consiste à prévoir un second bloc qui, lui, sera exécuté si l'expression est fausse. La question traitée change donc quelque peu de nature : il ne s'agit plus de savoir s'il faut agir ou non, mais de décider laquelle de deux actions doit être effectuée.

```
if (expression)
    bloc 1
else
    bloc 2
```

Un bloc ne comportant qu'une instruction peut, là aussi, être débarrassé de ses accolades. Le risque d'erreur est encore plus grand dans ce cas que dans le précédent.

```
//exemple d'exécution alternative
//(a et b sont des variables numériques supposées précédemment définies)
1 if(a > b) //remarquez l'absence de point virgule sur cette ligne
2 {
3     a = b; //ces instructions ne seront exécutées que si le contenu de a
4     b = 4; //est plus grand que celui de b
5 }
6 else      //il n'y a pas non plus de point virgule ici
7 {
8     b = a; //ces instructions ne seront exécutées que si le contenu de a
9     a = 9; //est inférieur ou égal à celui de b
10 }
11 b = 2 * b ; //cette instruction est exécutée dans tous les cas
```

² Si l'omission des accolades non indispensables est systématique dans les exemples contenus dans les Leçons, ce n'est que pour vous faire économiser du papier et faciliter la lecture du texte en limitant la fréquence des sauts de page interrompant un fragment de code. Il ne s'agit en aucun cas d'un style à encourager dans le code réel.

Expression alternative

Dans certains cas, l'effet d'une exécution alternative peut être obtenu plus directement au moyen d'une *expression* alternative, qui utilise le seul **opérateur ternaire** du langage C++, noté "**? :**". Cet opérateur implique un premier opérande booléen, alors que les deux autres opérandes peuvent être d'un type quelconque. La valeur de l'expression dépend directement de celle du **premier opérande** : s'il est vrai, l'expression a la même valeur que son **deuxième opérande**, s'il est faux, l'expression a la même valeur que son **troisième opérande**.

```
1 //exemple de calcul utilisant l'opérateur ternaire
2 double maxi; //(a et b sont des variables numériques supposées précédemment définies)
3 maxi = (a > b) ? a : b;
4
5 //le même effet pourrait être obtenu par exécution alternative :
6 if (a > b)
7     maxi = a;
8 else
9     maxi = b;
```

L'avantage de l'*expression* alternative sur une *exécution* alternative est, s'il existe, purement subjectif. On peut, sans doute, trouver que l'intention principale (déterminer la valeur de maxi) est plus apparente dans le premier cas que dans le second, mais le code généré par le compilateur a toutes les chances d'être strictement le même dans les deux cas.

L'usage de l'expression alternative n'est pas limité aux cas où le *résultat* de son évaluation est utilisé. Il est, dans tous les cas, absolument garanti qu'un seul des deux derniers opérandes sera évalué, ce qui signifie que, si l'évaluation de chacun d'entre eux a un *effet* différent, la valeur du premier opérande (booléen) détermine lequel de ces effets sera obtenu.

```
//exemple d'utilisation de l'EFFET de l'évaluation d'une expression utilisant
//l'opérateur ternaire (f1() et f2() sont deux fonctions supposées définies)
//(a et b sont des variables numériques supposées précédemment définies)
(a > b) ? f1() : f2() //une seule des deux fonctions sera appelée
```

Exécution à choix multiple

Si l'exécution alternative permet de décider laquelle de deux actions doit être effectuée, on peut facilement imaginer des cas où le choix doit se faire entre plus de deux actions possibles. La décision ne peut alors plus être prise en fonction d'une valeur booléenne, puisque le type booléen n'offre que deux valeurs différentes.

Le langage C++ permet alors d'effectuer le choix en fonction d'une valeur d'un type entier ordinaire (char, short, int ou long) ou d'un type énuméré. La condition n'est plus introduite par le mot if, mais par le mot **switch** et les descriptions des différentes actions ne sont plus délimitées par le mot else mais par le mot **break**. L'association entre les valeurs et les actions est obtenue en annonçant, à l'aide du mot **case**, à quelle valeur correspond chaque description d'action. A ce principe de base s'ajoutent deux raffinements : on peut créer, à l'aide du mot **default**, un cas correspondant à tous ceux non cités par ailleurs, et l'omission du mot break permet d'avoir des portions de code qui sont exécutées dans plusieurs cas.

Une des singularités de la structure switch est que les séquences d'instructions associées aux différents case ne sont pas regroupées en blocs, mais simplement délimitées en amont par les deux points suivant le case et en aval par un break, ou par l'accolade fermante du switch. Ces séquences n'étant pas des blocs, il n'est pas possible d'y définir des variables.

La structure switch n'est pas des plus élégantes, car elle nécessite beaucoup de vocabulaire spécifique et présente des particularités de fonctionnement ouvrant la porte à de véritables abominations, dont le célèbre Duff's device (<http://www.lysator.liu.se/c/duffs-device.html>) n'est qu'un exemple extrême. Son usage est cependant assez répandu et ne peut pas vraiment être déconseillé : s'il est toujours possible de remplacer un switch par une succession de if, ce n'est pas toujours très facile (en particulier lorsque les raffinements susmentionnés sont utilisés). Si vous ne cherchez pas délibérément les ennuis, je vous suggère toutefois de vous en tenir à un usage "canonique" du switch, du genre de celui illustré par le TD 03 et par l'exemple suivant.

```
//exemple d'exécution à choix multiple
//(unCaractere est une variable de type char supposée précédemment définie)
1 switch(unCaractere)
2 {
3     case 'a' :
4         //les lignes placées ici ne seront exécutées que si unCaractere == 'a'
5         break;
6     case 'e' :
7         //les lignes placées ici ne seront exécutées que si unCaractere == 'e'
8         break;
9     case 'i' :
10        //les lignes placées ici ne seront exécutées que si unCaractere == 'i'
11    case 'o' :
12        //les lignes placées ici seront exécutées si unCaractere
13        //contient 'i' ou 'o'
14    case 'u' :
15        //les lignes placées ici seront exécutées si unCaractere
16        //contient 'i','o' ou 'u'
17    case 'y' :
18        //les lignes placées ici seront exécutées si unCaractere
19        //contient 'i','o', 'u' ou 'y'
20        break;
21    default :
22        //les lignes placées ici seront exécutées si unCaractere contient autre
23        //chose qu'une voyelle non accentuée
24        //le break précédant la fin du switch peut être omis
25 }
26 //les lignes placées ici seront exécutées dans tous les cas
```

3 - Itérations

Décider si un bloc d'instructions doit ou non être exécuté c'est, d'un certain point de vue, décider s'il doit être exécuté 1 ou 0 fois. Vue sous cet angle, l'exécution conditionnelle n'est qu'un cas particulier d'un problème plus général : contrôler *combien de fois* un bloc d'instructions doit être exécuté.

La boucle while()

Une première façon d'organiser l'itération est de commencer par vérifier si la condition de répétition est remplie et, si elle l'est, de répéter l'exécution du bloc d'instructions³ jusqu'à ce que la condition cesse d'être remplie. Cet effet est obtenu à l'aide du mot `while` précédant l'expression booléenne placée entre parenthèses :

```
while (expression)
    bloc
```

Cette construction laisse donc la possibilité que le bloc ne soit, en fait, jamais exécuté. C'est en effet ce qui se produit si la condition de répétition n'est pas vérifiée au moment de l'exécution de la ligne comportant le `while()`.

A titre d'exemple, le fragment de code suivant utilise une boucle `while()` pour faire la somme des nombres entiers positifs inférieurs à 5 :

```
//exemple d'utilisation d'une boucle while()
1 int somme = 0;
2 int n = 1;
3 while (n < 5)
4 {
5     somme = somme + n;
6     n = n + 1;
7 }
```

³ Comme dans le cas des structures d'exécution conditionnelle, le bloc peut être remplacé par une instruction unique (sans accolades), mais cette simplification est parfois source d'erreurs.

La boucle do ... while();

On peut aussi n'effectuer le test qu'après que le bloc d'instructions a été exécuté une première fois. Cet effet est obtenu à l'aide du mot `do` précédant le bloc d'instructions, lui-même suivi du mot `while` et de l'expression booléenne placée entre parenthèses.

```
do
    bloc
while(expression); //remarquez le point-virgule !
```

Cette construction est particulièrement adaptée aux cas où il n'y a pas de valeur significative à tester avant l'exécution du bloc (c'est le cas, par exemple, lorsqu'il s'agit de répéter une saisie de donnée tant que la valeur reçue n'est pas valide).

Le fragment de code suivant effectue, lui aussi, la somme des quatre premiers entiers positifs :

```
1 //exemple d'utilisation d'une boucle do ... while()
2 int somme = 0;
3 int n = 1;
4 do {
5     somme = somme + n;
6     n = n + 1;
7 } while (n < 5);
```

La boucle for(; ;)

Dans les deux structures précédentes, il est évidemment nécessaire que certaines des instructions présentes dans le bloc aient un effet sur l'expression qui contrôle la répétition, sans quoi la répétition de l'exécution du bloc ne s'achèvera jamais. Les syntaxes du `while()` et du `do ... while();` ne rendent pas particulièrement visibles ces instructions, ce qui nuit parfois à la lisibilité du code.

Le langage C++ propose une troisième structure itérative, qui invite à rendre le contrôle de la condition de répétition plus explicite. Cet effet est obtenu à l'aide du mot `for` accompagné d'un couple de parenthèses incluant trois expressions séparées par des points virgules, l'ensemble précédant le bloc d'instructions dont l'exécution doit être répétée. Le rôle exact des trois expressions est facilement décrit en disant que

```
for (expression1 ; expression2 ; expression3)
    bloc
```

est exactement équivalent à

```
expression1;
while(expression2)
{
    bloc
    expression3;
}
```

En clair, `expression2` est (a priori) une expression booléenne qui contrôle la répétition du bloc, alors que `expression3` a vocation à modifier la valeur de `expression2`. Quant à `expression1`, son évaluation préalable à l'entrée dans le cycle de répétitions en fait un candidat naturel pour l'établissement des valeurs initiales de certaines variables.

```
1 //exemple d'utilisation d'une boucle for( ; ; )
2 int n;
3 int somme = 0;
4 for(n = 1 ; n < 5 ; n = n + 1) //remarquez l'absence de point virgule
5 {
6     somme = somme + n;
7 }
```

La syntaxe du `for(; ;)` exige seulement la présence des parenthèses et des deux points virgules : chacune des trois expressions peut être ou non présente, et leur rôle exact est laissé à l'initiative du programmeur, ce qui conduit souvent à des formes "dégradées" de cette boucle, dont la lisibilité est parfois assez douteuse. Lorsque l'évaluation d'au moins une des expressions utilisées dans le `for(; ;)` crée un *effet*, il arrive que le bloc d'instructions à

répéter se trouve lui-même omis. Ce bloc est alors remplacé par un point virgule concluant la ligne du `for(; ;)`, ce qui est encore plus déconcertant pour le lecteur novice.

Conversion des structures itératives les unes en les autres

Il convient d'être conscient du fait que les trois formes de boucles proposées par le langage sont parfaitement réductibles les unes aux autres. En fait, comme le montre le tableau d'équivalences présenté ci-dessous, le `while()` est facilement réductible à un `for(; ;)`, alors que la transformation d'un `do ... while();` en une autre forme d'itération implique la duplication du bloc dont l'exécution doit être répétée⁴. La transformation d'un `for(; ;)` en une boucle `while()` est assez directe, mais présente l'inconvénient de "noyer" la troisième expression dans le bloc d'instructions à répéter, ce qui risque de rendre plus difficile la perception de la logique du programme.

Forme canonique	Formes équivalentes		
	<code>while()</code>	<code>do ... while();</code>	<code>for(...;...;...)</code>
<code>while (expr) bloc</code>		<code>if(expr) do bloc while (expr);</code>	<code>for(; expr ;) bloc</code>
<code>do bloc while(expr);</code>	<code>bloc while(expr) bloc</code>		<code>bloc for(; expr ;) bloc</code>
<code>for (expr1; expr2; expr3) bloc</code>	<code>expr1; while(expr2) { bloc expr3; }</code>	<code>expr1; if(expr2) do { bloc expr3; } while(expr2);</code>	

Le choix d'un type de boucle plutôt que d'un autre est donc largement une question d'évaluation subjective de la lisibilité du programme.

4 - Bon, c'est gentil tout ça, mais ça fait déjà 7 pages. Qu'est-ce que je dois vraiment en retenir ?

- 1) On peut former une expression booléenne en comparant deux termes à l'aide d'un des opérateurs de comparaison.
- 2) Les expressions booléennes peuvent faire l'objet de calculs logiques utilisant les opérateurs `&&` (qui signifie ET), `||` (qui signifie OU) et `!` (qui signifie NON).
- 3) Une expression booléenne permet de décider si un bloc de code doit ou non être exécuté. (A l'aide du mot `if`).
- 4) Une expression booléenne permet de décider lequel de deux blocs de code doit être exécuté. (A l'aide des mots `if` et `else`).
- 5) Une expression dont la valeur est d'un type entier ou énuméré permet de décider lequel de plusieurs groupes d'instructions doit être exécuté. Cet effet est obtenu à l'aide des mots `switch`, `case`, `break` et `default`.
- 6) Le langage C++ contrôle toujours la répétition de l'exécution d'un bloc d'instructions à l'aide d'une *condition de répétition*, c'est à dire d'une expression booléenne qui doit devenir fausse lorsqu'il faut s'arrêter⁵.
- 7) La structure `do ... while();` permet d'exécuter une fois le traitement avant de se demander s'il est nécessaire de continuer.
- 8) Les structures `while()` et `for(; ;)` permettent de vérifier si le traitement est nécessaire avant de l'entreprendre.
- 9) Mis à part la nuance soulignée par les points 7 et 8, les trois formes d'itérations proposées par C++ sont essentiellement équivalentes.

⁴ Tout simplement parce que `do ... while();` est la seule boucle qui exécute le bloc d'instructions AVANT le test.

⁵ Par opposition à une condition d'arrêt, qui deviendrait vraie quand il faut s'arrêter.