Faculté Informatique & Communications
Cours d'informatique

Mini-Référence :

Structures de contrôle en C++

Bloc d'instructions (instructions composées)

En C++, un *bloc* est un groupe d'instructions délimité par { et } :

```
{
    // Ceci est un bloc
    int i(0);
    i = i + 1;
    ...
}
```

Branchement conditionnel «si ... sinon» : if ... else

Syntaxe :	**if (** «*condition*» **) {** «*actions si*» **} else {** «*actions sinon*» **}**

où:

- «*condition*» est une *expression logique*,
- «*actions si*» est la séquence d'instructions à exécuter si la condition est vérifiée,
- «*actions sinon*» est la séquence d'instructions à exécuter si la condition n'est pas vérifiée.

Remarques :

Il n'y a pas de point-virgule à la fin du bloc de traitements du branchement conditionnel.

Attention à ne pas confondre l'opérateur d'**affectation** = avec l'opérateur relationnel de **test d'égalité** == utilisé pour former les conditions logiques.

Sélection à choix multiple : switch

Syntaxe :	**switch (**«*expression*»**) {** **case** «cst_1» **:** «$actions_1$»**;** ... **case** «cst_n» **:** «$actions_n$»**;** **default :** «*actions par défaut*»**;**

	}

où:

- «*expression*» est une *expression entière*, *booléenne* ou *caractère*.
- «*cst_i*» est une expression constante, i.e. constituée de littéraux et de constantes de **même type** que «*expression*
  (et par conséquent évaluable lors de la compilation)
- «*actions par défaut*» est la séquence (optionelle) d'instructions à exécuter dans le cas où il n'y a pas de *cst_i* évalué égale au résultat de l'expression de sélection.

Remarques :

- Il n'y a pas de point-virgule à la fin du bloc de traitements du branchement conditionnel ;
- «*$actions_i$*» peut être une suite d'instructions terminées chacune par un point-virgule ;
- Pour terminer une séquence d'actions liée à une clause case, il faut utiliser l'instruction de rupture de séquence break;.

Exemple:

```
int a(5), b(1);
...
switch (a-b) {
    case 0  :
    case 1  : cout << "a est proche de b" << endl;
              break;
    case 3  :
    case 5  : cout << "'a-b' n'est pas une valeur binaire..." << endl;
              break;
    default : cout << "'a-b' est pair ou est plus grand que 6" <<
endl;
}
```

Les boucles «*Tant que ...* » : while

Syntaxe :	**while (**«*condition*»**) {**«*actions*»**}** Il n'y a pas de point-virgule après le bloc {«*actions*»}.

Lorsque le programme passe par la structure de contrôle **while**, les actions suivantes sont réalisées :

1. évaluation de l'expression «*condition*»,
2. si la *condition* est **vérifiée**, exécution du bloc «*actions*» et retour en (1).

Dans le cas où la condition est **fausse** (i.e. pas vérifiée), l'exécution se poursuit immédiatement après le bloc «*actions*», sans entrer dans celui-ci.

Avec la structure *while*, **la condition de continuation est évaluée avant l'exécution des actions**.
C'est en fait une *condition d'entrée* dans la structure de boucle.

Exemple : saisie d'une valeur dans un intervalle

```
int n(1);
cout << "Saisie de la valeur..."
<< endl;
while ( (n < 1) || (n > 3) ) {
    cout << "Entrez une valeur
entre 1 et 3:";
    cin >> n;
}
cout << "La valeur entrée est: "
<< n << endl;
```

Résultat :

```
Saisie de la valeur...
La valeur entrée est: 1
```

Le résultat n'est pas celui attendu... la boucle n'est même pas exécutée une fois.
Pour entrer effectivement dans la boucle, il faudrait dans ce cas utiliser une valeur d'initialisation de n hors de l'intervalle (par exemple, avec int n(0);).

Les boucles «*Faire ... tant que ...* » : do ... while

Syntaxe : **do {**«*actions*»**} while (**«*condition*»**);**

Il y a un point-virgule après l'expression («*condition*»)

Lorsque le programme passe par la structure de contrôle **do ... while**, les actions suivantes sont réalisées:

1. exécution du bloc «*actions*»,
2. évaluation de l'expression «*condition*»,
3. Si la condition est **vérifiée**, retour en (1).

Dans le cas où la condition est **fausse** (i.e. pas vérifiée), l'exécution se poursuit avec l'instruction suivant immédiatement l'expression «*condition*».

Avec la structure *do ... while*, **la condition de continuation est évaluée après l'exécution des actions**.

Exemple : saisie d'une valeur dans un intervalle

```
int n;
cout << "Saisie de la valeur..."
<< endl;
do {
    cout << "Entrez une valeur
entre 1 et 3:";
    cin >> n;
} while ( (n < 1) || (n > 3) ) ;
cout << "La valeur entrée est: "
```

Résultat:

```
Saisie de la valeur...
Entrez une valeur entre 1 et 3: 5
Entrez une valeur entre 1 et 3: 0
Entrez une valeur entre 1 et 3: 3
La valeur entrée est: 3
```

Dans ce cas, quelque soit la valeur d'initialisation de n, le programme passe au moins 1 fois par la boucle, et la saisie de la valeur est effectuée.

```
<< n << endl;
```

Les itérations : for

Syntaxe :	**for (**«*initialisation*» **;** «*condition*» **;** «*mise à jour*» **) {** «*actions*» **}**

Il n'y a pas de point-virgule après le bloc d'actions.

Lorsque le programme passe par la structure itérative **for**, les actions suivantes sont réalisées :

1. exécution de l'«*initialisation*»,
2. évaluation de la «*condition*»,
3. si la condition est **réalisée**, exécution de bloc «*actions*» (ce qui correspond à une *itération*), puis exécution des instructions «*mise à jour*» et retour en 2.
4. si la condition n'est **pas réalisée**, sortie de la structure itérative (l'exécution se poursuit avec l'instruction suivant le crochet fermant de la structure)

Exemple :

```
for (int n(0); n < 2; ++n) {
    cout << "n = " << n << endl;
}
```

Résultat :

```
n = 0
n = 1
```

La rupture de séquence : break

Syntaxe :	**break;**

L'instruction break; permet d'interrompre prématurément le déroulement d'une boucle, en sautant toutes les actions (de la structure) qui la suivent; il n'y a par ailleurs pas de test de la condition de continuation.

Exemple :

```
while (true) {
  cout << "Entrez une valeur: "
<< flush;
  cin >> value;
  /*
   * On test la validité de la
valeur entrée
   * ([2,12]), si la valeur est
bonne
   * (i.e. dans le domaine), on
poursuit
   * l'exécution, sinon on
affiche un message
   * d'erreur et on recommence la
```

Résultat :

```
Entrez une valeur: 4
Valeur ok !
```

```
saisie....
  */
  if ((value >= 2) && (value <=
12)) {
      break;
  }
  cout << endl
       << "La valeur DOIT être
dans l'intervalle ["
       << 2 << ',' << 12 << ']'
<< endl;
}
cout << "Valeur ok !" << endl; //
<-- break mène ici.
```

Remarques :

- Evitez au maximum d'utiliser des boucles infinies comme ci-dessus (while (true)), explicitez toujours les conditions de sortie de la boucle.
- Il n'y a pas de point-virgule à la fin du bloc de traitements du branchement conditionnel.

La rupture de séquence : continue

Syntaxe :	**continue;**

Lorsque le programme rencontre l'instruction continue (à l'intérieur d'un bloc *actions* d'une boucle), il passe directement à la fin de ce bloc, sans exécuter les instructions qui le suivent. Cependant, il ne **sort pas** de la boucle, contrairement à break.

Exemple :

```
for (int i(0); i < 8; ++i) {
  int j = i*2+3;

  /* si j < 10: on passe
directement à l'itération
   * suivante (prochaine valeur
de i)
   */
  if (j < 10) continue;

  // sinon, on passe ici.
  cout << i << ", " << j << endl;
}
```

Résultat :

```
4, 11
5, 13
6, 15
7, 17
```

Expressions logiques : évaluation paresseuse

Les opérateurs logiques && et || effectuent une **évaluation paresseuse** de leur arguments :

- l'évaluation des arguments se fait de la gauche vers la droite;
- ne sont évalués que les arguments strictement nécessaires à la détermination de la valeur logique de l'expression conditionnelle.

Par exemple, dans l'expression conditionnelle «(a && b && c && ...)», si a est faux, cela assure que toute l'expression est fausse, et les autres arguments ne sont pas évalués (les arguments sont évalués *jusqu'au 1er argument faux*).
De la même façon, dans l'expression conditionnelle «(a || b || c || ...)», si a est vrai, cela assure que toute l'expression est vraie, et les autres arguments ne sont pas évalués (les arguments sont évalués *jusqu'au 1er argument vrai*).

Exemples :

```
( (x != 0) && (4/x > 3) )
```

L'expression 4/x produirait une erreur si xétait égal à zéro au moment de son évaluation. Mais du fait de l'évaluation paresseuse, cette expression n'est pas calculée quand x=0, et ne produit pas d'erreur. En revanche, la condition suivante produit une erreur pour x=0 :

```
( (4/x > 3) && (x != 0) )
```

On peut également créer des expressions conditionnelles assez longues sans pour autant que leur évaluation soit forcément coûteuse (en terme de temps de calcul).
L'expression conditionnelle :

```
(ch=='a' || ch=='e' || ch=='i' || ch=='o' || ch=='u'|| ch=='y')
```

qui teste si le caractère ch est une voyelle ne sera évaluée complètement que si ch est effectivement une consonne, ou la voyelle y. (On a donc intérêt à tester en premier les conditions les plus probables, soit pour notre exemple et en admettant que le caractère testé soit issu d'un mot en langue latine, ch == 'e').
Notez que dans ce dernier exemple, on aurait plutôt intérêt à écrire un switch(ch) plutôt qu'un gros «ou» comme fait ici.

Dernière mise à jour : 2005/11/09 15:06:12 (1.8)