Structures conditionnelles [if]
Support de Cours

Karine Zampieri, Stéphane Rivière

Unisciel algoprog Version 14 mai 2018

Table des matières

C++ - Structures conditionnelles (Cours)

Mots-Clés Conditions, Sélective Si, Sélective Si-Sinon-Si, Sélective Selon ■
Requis Qu'est-ce qu'un algorithme, Structures de base ■
Difficulté • ○ ○ (3 h) ■

Introduction

Ce module traite des notions de **conditions** (simples, booléens, conditions composées) puis introduit les **structures conditionnelles** (`Si`, `Si-Alors` et `Si-Sinon-Si`) et définit l'arbre de choix . Le *Compléments* décrit la sélective `Selon` ainsi que l'opérateur `Si-`expression.

1 Conditions

1.1 Opérateurs de comparaison

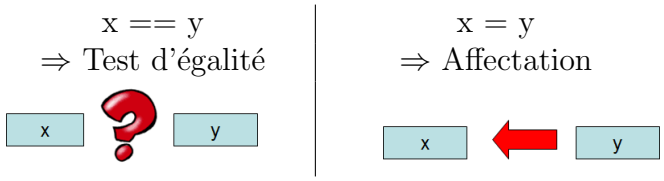
Opérateurs de comparaison

Dits aussi **opérateurs relationnels** ou **comparateurs**, ils agissent généralement sur des variables numériques ou des chaînes et donnent un résultat booléen. Pour les caractères et chaînes, c'est l'ordre alphabétique qui détermine le résultat.

C/C++ **Opérateurs de comparaison**

Opérateur Mathématique	Signification	Équivalent Algorithmique	Équivalent C/C++
$<$	(strictement) inférieur	`a < b`	`a < b`
$\leq$	inférieur ou égal	`a <= b`	`a <= b`
$>$	(strictement) supérieur	`a > b`	`a > b`
$\geq$	supérieur ou égal	`a >= b`	`a >= b`
$=$	égalité	`a = b`	`a == b`
$\neq$	différent de (ou inégalité)	`a <> b`	`a != b`

Distinguer == et =



A gauche Compare la valeur de x à celle de y et rend true si elles sont égales, false sinon (et donc **ne modifie pas** la valeur de x)

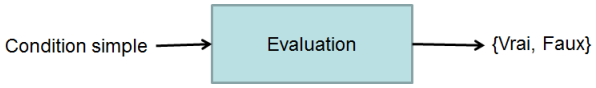
A droite Affecte à la variable x la valeur de y (et donc **modifie** la valeur de x)

1.2 Condition simple

Condition simple

Notée $v_1 \ \Phi \ v_2$, elle associe un *opérateur de comparaison* Φ et deux valeurs v_1 et v_2 de même nature (même type ou types comparables) et délivre un résultat booléen :

- `Vrai` : on dit que la condition est vérifiée.
- `Faux` : cas de condition non vérifiée.



Exemples

- `eval(2<3)` est `Vrai`.
- `eval(6*3=13)` est `Faux`.
- `eval('A'<'E')` est `Vrai` car l'ordre alphabétique est respecté dans les codes ASCII attribués aux lettres.
- `eval("milou"<"tintin")` est `Vrai` de même que `eval("assembleur"<="java")`.
- `eval('a'<'A')` est `Faux` car les majuscules sont avant les minuscules.
- `eval("Bonjour">"Bon jour")` est `Faux` car l'espace est avant les lettres.
- Si `n1` et `n2` sont deux variables de type entier contenant les valeurs 7 et 5, `eval(2*n1>n2+3)` est `eval(2*7>5+3)` c.-à-d. `eval(14>8)` donc `Vrai`.

Tester x dans (a..b)

Contrairement aux mathématiques, les opérateurs de comparaison **ne peuvent pas** être enchaînés. Ainsi, pour vérifier si un nombre $x \in [a..b]$, il faut écrire :

$$(a <= x \text{ Et } x <= b)$$

En effet :

$$\begin{aligned} & (a <= x <= b) \\ \equiv & ((a <= x) <= b) \\ \equiv & (\{\text{Faux Ou Vrai}\} <= b) \end{aligned}$$

On compare donc un booléen à un entier avec l'opérateur inférieur ($<$) : erreur, les types sont différents. Cependant dans les langages comme le C/C++, le Faux équivaut au 0 et le Vrai à 1 : ici c'est donc vrai... pour tout b positif.
Dans tous les cas, il faudra exploiter les opérateurs logiques pour exprimer de telles conditions.

1.3 Opérateurs logiques

Opérateurs logiques

Dits aussi **connecteurs logiques** ou **opérateurs booléens**, ils agissent sur des expressions booléennes (variables ou expressions à valeurs booléennes) pour donner un résultat du même type. Ils peuvent être enchaînés.

Opérateurs logiques

Opérateur Mathématique	Signification	Équivalent Algorithmique	Équivalent C/C++
¬	négation (unaire)	`Non a`	`!a`
∧	conjonction logique	`a Et b`	`a && b`
∨	disjonction logique (ou inclusif)	`a Ou b`	`a || b`

Opérateurs logiques

Opérateur Mathématique	Signification	Équivalent Algorithmique	Équivalent C++
¬	négation (unaire)	`Non a`	`not a`
∧	conjonction logique	`a Et b`	`a and b`
∨	disjonction logique (ou inclusif)	`a Ou b`	`a or b`

Opérateur Ou-exclusif

Il n'y a pas d'opérateur OU-exclusif (`xor`) logique.

1.4 Propriétés des opérateurs logiques

Propriétés des opérateurs logiques

Elles sont définies sous forme de tableaux communément appelés **tables de vérité**. Ces tableaux se lisent ligne par ligne.

x	y	¬x (non x)	x ∧ y (x Et y)	x ∨ y (x Ou y)
Vrai	Vrai	F	Vrai	Vrai
Vrai	F		F	Vrai
F	Vrai	Vrai	F	Vrai
F	F		F	F
F = Faux				

- `c1 Et c2` est vrai ssi les deux conditions sont vraies.
- `c1 Ou c2` est faux ssi les deux conditions sont fausses.

Variable booléenne

Pour un booléen `b` :

- `b = Faux` est équivalent à `Non b`
- `b = Vrai` est équivalent à `b`
- `Non Non b` est équivalent à `b`

Dans les trois cas, nous préconiserons la seconde écriture.

1.5 Condition composée

Condition composée
(Plus simplement **condition** ou **expression logique**) Notée $b_1\ \Psi_1\ b_2\ \Psi_2 \ldots$, elle associe des *opérateurs logiques* Ψ_i et des valeurs booléennes b_j et délivre un résultat booléen (`Vrai` ou `Faux`). Elle permet de relier des conditions simples en une seule « super-condition ».

Exemples

- L'entier `a` doit être strictement supérieur à zéro et strictement inférieur à 100 :

```
a > 0 && a < 100
a > 0 and a < 100
```

- La couleur `c` doit être `Rouge`, `Verte` ou <`Bleue` :

```
c == Rouge || c == Vert || c == Bleu
c == Rouge or c == Vert or c == Bleu
```

- La couleur `c` ne doit pas être `Noire` :

```
!(c == Noir)
not(c == Noir)
```

- « A un feu de croisement, je m'arrête s'il est rouge ou s'il est orange et si ma vitesse est inférieure à 40km/h. Dans les autres cas, je passe. »

```
arret = (feu == rouge) or (feu == orange and vitesse < 40)
passe = not arret
```

Théorèmes de De Morgan
Ils énoncent :

$$\begin{aligned} \text{Non(a Et b)} &\Leftrightarrow \text{Non a Ou Non b} \\ \text{Non(a Ou b)} &\Leftrightarrow \text{Non a Et Non b} \end{aligned}$$

Utilité
Formulés par le mathématicien britannique, Augustus De Morgan, les théorèmes énoncent des équivalences entre des expressions booléennes faisant intervenir des négations et permettent de les simplifier.

Résumé des propriétés
Les opérateurs vérifient les propriétés suivantes :

$$\begin{aligned} \neg\neg a &= a \\ \neg(a \wedge b) &= \neg a \vee \neg b \\ \neg(a \vee b) &= \neg a \wedge \neg b \\ a \wedge (b \vee c) &= (a \wedge b) \vee (a \wedge c) \\ a \vee (b \wedge c) &= (a \vee b) \wedge (a \vee c) \end{aligned}$$

1.6 Priorité des opérateurs

C/C++ : Priorité des opérateurs

Les opérateurs de même priorité sont regroupés sur une même ligne.

Priorité	Opérateur Algorithmique	Opérateur C/C++	Signification
La plus élevée	- (unaire)	- (unaire)	Négation algébrique
	^	(aucun)	Puissance
	* / div mod	* / / %	Multiplication, division, div. entière, modulo
	+ -	+ -	Addition et soustraction
	< <= > >=	< <= > >=	Opérateurs de comparaison
	= <>	== !=	Opérateurs d'égalité
	Non	!	Négation logique
	Et	&&	Et logique
La plus basse	Ou	||	Ou logique

Cas de combinaisons de Et et de Ou

Mettez des parenthèses :

```
(cond1 Et cond2) Ou cond3
```

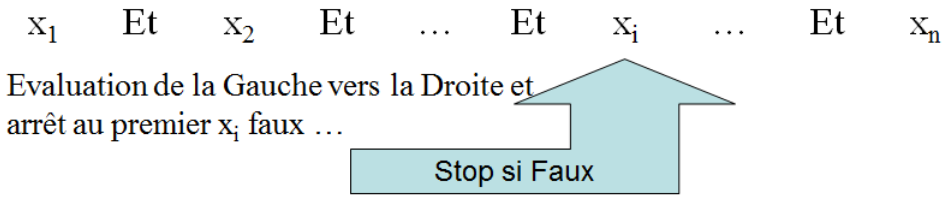
est différent de

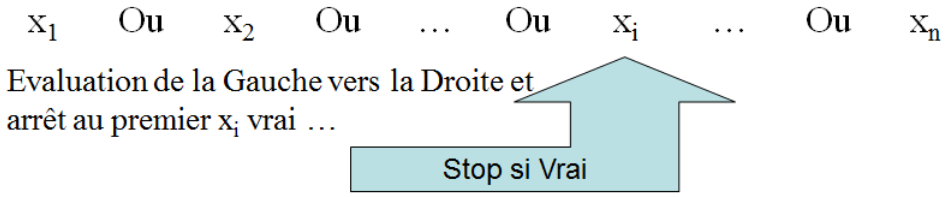
```
cond1 Et (cond2 Ou cond3)
```

En l'absence de parenthèses, le `Et` est prioritaire sur le `Ou`.

1.7 Évaluation paresseuse des opérateurs Et et Ou

Principe de l'évaluation paresseuse
(« *Lazy evaluation* ») Dite aussi **évaluation court-circuitée** (« *shortcut* »), elle s'effectue de la **gauche vers la droite** et ne sont évalués que les conditions **strictement nécessaires** à la détermination de la valeur logique de l'expression.





Utilité de l'évaluation paresseuse
Elle permet de gagner du temps mais surtout elle évite des erreurs d'exécution.

Exemple
considérons l'expression :

```
n <> 0 Et m/n > 10
```

Si `n` est nul, l'évaluation paresseuse donne le résultat `Faux` immédiatement après test de la première condition sans évaluer la seconde, tandis qu'une évaluation complète entrainerait un arrêt de l'algorithme pour cause de division par 0.

Non-commutativité du Et et du Ou
L'évaluation paresseuse a pour conséquence :

`c1 Et c2`
n'est pas équivalent à
`c2 Et c1`

1.8 Exemples : Priorité et évaluation

Exemple : Priorité des opérateurs

Considérons l'expression logique :

```
non a+2<30 ou (b-c/2=28 et c^11>2000*c+1)
```

Elle est équivalente à :

```
(non((a+2)<30))
ou (((b-(c/2))=28) et ((c^11)>((2000*c)+1)))
```

La formule avec sous-accolades est :

$$\underbrace{(\text{non}\,\underbrace{(\underbrace{(a+2)}_{}<30)}_{})}_{}\ \text{ou}\ \underbrace{\underbrace{(\underbrace{(b-\underbrace{(c/2)}_{})}_{}=28)}_{}\ \text{et}\ \underbrace{(\underbrace{(c^{11})}_{}>\underbrace{(\underbrace{(2000*c)}_{}+1)}_{})}_{}}_{}$$

Exemple : Évaluation paresseuse

Le tableau ci-dessous rassemble les évaluations de l'expression booléenne :

```
expr <-- (A > 10 et A <= 12) ou (non (B > 10))
```

Par exemple, la deuxième colonne du tableau montre que si `A` contient 0 et `B` contient -3, alors l'évaluation booléenne de `expr` donne `Vrai`. De plus, si la case est vide, ceci signifie que la condition n'est pas évaluée en raison de l'évaluation paresseuse.

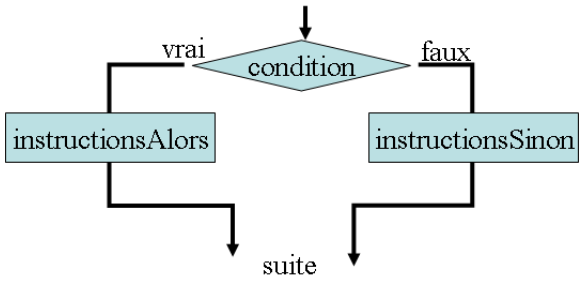
A	0	11	13
B	-3	16	16
A > 10	Faux	Vrai	Vrai
A <= 12		Vrai	Faux
A > 10 et A <= 12	Faux	Vrai	Faux
B > 10	Faux		Vrai
non (B > 10)	Vrai		Faux
expr <- (A > 10 et A <= 12) ou (non (B > 10))	Vrai	Vrai	Faux

2 Sélectives Si, Si-Alors et Si-Sinon-Si

2.1 Sélective Si

Sélective Si

Elle traduit : `Si` la `condition` est vraie, exécuter les `instructionsAlors`, `Sinon` exécuter les `instructionsSinon`. Il s'agit d'un choix binaire : **une et une seule** des deux séquences est exécutée.



La `condition` peut être simple ou complexe (avec des parenthèses et/ou des opérateurs logiques `Et`, `Ou`, `Non`).

C/C++

Sélective Si

```
if (condition)
{
  instructionsAlors;
}
else
{
  instructionsSinon;
}
```

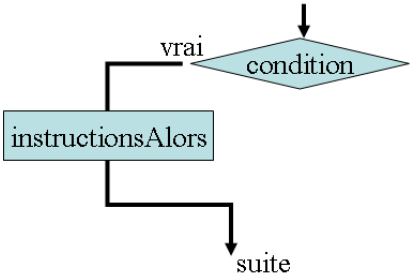
C/C++

Notez l'absence du mot-clé `Alors` d'où l'obligation des parenthèses autour de la condition.

2.2 Sélective Si-Alors

Sélective Si-Alors

Forme restreinte de la structure `Si` (sans clause `Sinon`).



C/C++ **Sélective Si-Alors**

```
if (condition)
{
  instructionsAlors;
}
```

2.3 Remarques

Syntaxes Sélection binaire et blocs

Utilisez la syntaxe avec des blocs (même s'il n'y a qu'une seule instruction) :

- Ceci évite de retenir deux syntaxes différentes.
- En cas d'ajout d'instructions, le bloc est déjà présent.

C/C++ : L'expression logique !

Le langage considère comme expression logique une expression de n'importe quel type via la convention :

si eval(expression) est nulle
=> condition fausse
(sinon elle est vraie)

Les valeurs nulles sont : les zéros numériques (0 et 0.0), la valeur `false` et la valeur `void` (cf. @[Structuration de l'information, le type Pointeur]).

C/C++ : Que signifie « if (x)... » ?

Si x booléen	Si x non booléen
⇒ OK bonne écriture	⇒ Équivaut if (x != 0)

Conseil : Préférez l'écriture explicite des expressions logiques.

A propos des tests numériques

Puisque $2+3=5$, on a $\left(\sqrt{2}\right)^2+\left(\sqrt{3}\right)^2=\left(\sqrt{5}\right)^2$.
Mais que se passe-t-il si on utilise un tel calcul dans un test ? Le calcul est considéré comme faux car les réels-machine sont des valeurs approchées. Il convient donc d'être très prudent quand on utilise des égalités numériques (sur les réels) dans les tests.

2.4 Exemple : Sélectives Si et Si-Alors

Cet exemple saisit la moyenne annuelle d'un étudiant dans un entier `moy` puis affiche l'un des deux messages suivants :

```
Vous ne passez pas dans l'année supérieure
Bravo, vous passez dans l'année supérieure
```

Le premier est affiché si `moy < 10`, sinon c'est le deuxième. Et dans le cas où `16 <= moy <= 20`, il affiche également :

```
Avec les félicitations du jury
```

Programme @[pgpassage1.cpp]

```cpp
#include <iostream>
using namespace std;

int main()
{
  double moy;
  cout<<"Moyenne? ";
  cin>>moy;
  if (moy < 10.0)
  {
    cout<<"Vous ne passez pas dans l'annee superieure"<<endl;
  }
  else
  {
    cout<<"Vous passez dans l'annee superieure"<<endl;
    if (16.0 <= moy && moy <= 20.0)
    {
      cout<<"Avec les felicitations du jury"<<endl;
    }
  }
}
```

Explication

La première condition est une condition simple : `moy < 10`, et la deuxième condition une condition composée : `16 <= moy Et moy <= 20`.

Trace d'exécution

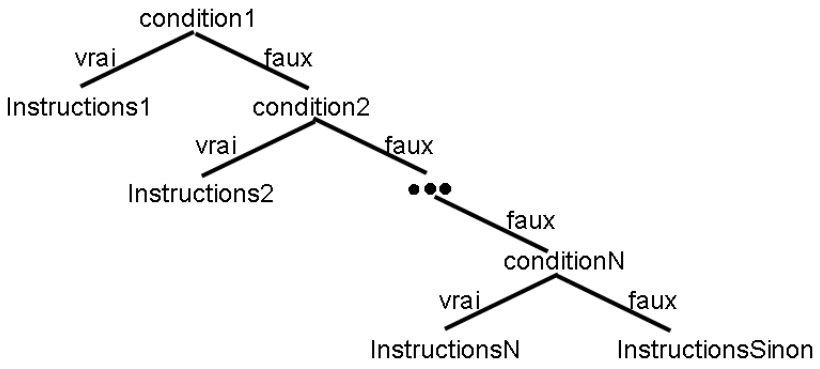
La présence d'une sélective `Si` ne change pas le principe de la trace d'exécution qui est de suivre pas à pas le contenu des variables.

Pour l'exemple, on a : `13 ==> "Vous passez dans l'année supérieure"`. En effet, après l'exécution de l'instruction de saisie `saisir(moy)`, la variable `moy` contient la valeur 13 et l'exécution commence à traiter la deuxième instruction. Cette deuxième instruction est une instruction `Si` mais comme sa condition `moy < 10` est fausse, le bloc d'instructions exécuté est celui de la clause `Sinon`...

2.5 Sélective Si-Sinon-Si

Sélective Si-Sinon-Si

Elle évalue successivement la `conditionI` et exécute les `instructionsI` si elle est vérifiée. En cas d'échec des `n` conditions, exécute les `instructionsSinon`.



C/C++ **Sélective Si-Sinon-Si**

```
if (condition1)
{
  instructionsA1;
}
else if (condition2)
{
  instructionsA2;
}
else if...
...
else if (conditionN)
{
  instructionsAn;
}
else
{
  instructionsSinon;
}
```

3 Arbre de choix

Si imbriquées et Si en cascade

Dans le cas de choix arborescents – un choix étant fait, d'autres choix sont à faire, et ainsi de suite –, il est possible de placer des structures `Si` de deux façons :

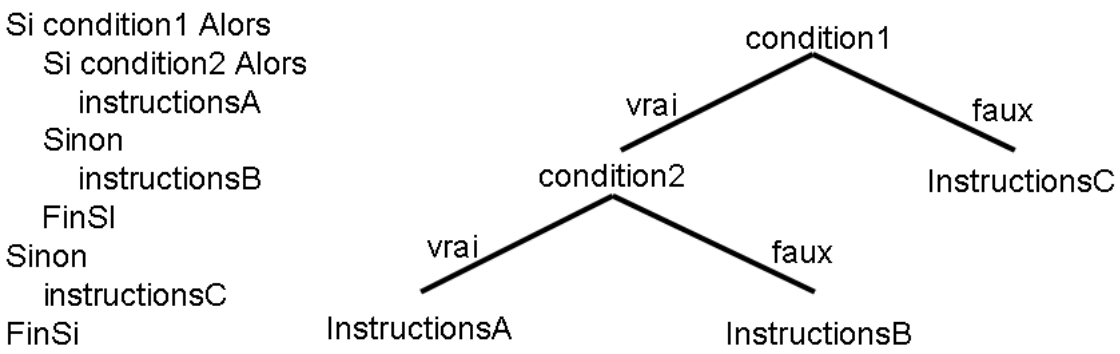
- **Si imbriquées** : A l'intérieur de chacune des clauses `Alors` et `Sinon`.
- **Si en cascade** : Uniquement dans la clause `Sinon`, d'où l'emploi du `Si-Sinon-Si`.

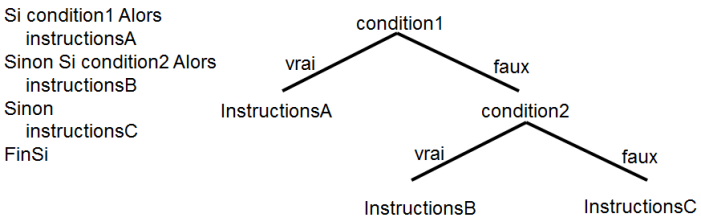
Il n'y a pas de limitation dans la profondeur des imbrications et/ou cascade sauf par rapport à la lisibilité de l'algorithme. L'unique règle à respecter est qu'à chaque `Si` doit correspondre au même niveau un `FinSi` et réciproquement. L'indentation est un moyen d'éviter ce type d'erreur.

Arbre de choix

Dit aussi **arbre de décision**, il permet de visualiser graphiquement les structures `Si`. La forme de l'arbre décrit immédiatement s'il s'agit de structures `Si` imbriquées et/ou en cascade.

Structures Si et son arbre de choix





Indentation

Dans l'écriture de tout programme, on veillera à **indenter** correctement les lignes de codes afin de faciliter sa lecture. Cela veut dire :

1. Les **balises** encadrant toute structure de contrôle devront être parfaitement à la verticale l'une de l'autre : `Début` avec `Fin` ; `Si`[, `Sinon`] avec `Finsi` ; (c'est vrai aussi pour celles que nous allons voir plus tard : `Selon` ; `TantQue` ; `Répéter` avec `Jusquà` ; `Pour` avec `FinPour`.

2. Les lignes situées entre toute paire de balises devront être décalées d'une tabulation vers la droite.
3. Sur papier, on tracera une **ligne verticale** entre le début et la fin d'une structure de contrôle afin de mieux la délimiter.

4 Compléments

4.1 Sélective Selon

La structure `Selon` est une simplification d'écriture de plusieurs alternatives imbriquées. Deux formes existent :

- Sélective `Selon` (listes de valeurs)
- Sélective `Selon` (conditions)

La forme acceptée par la plupart des langages de programmation est celle avec listes de valeurs.

Sélective Selon (listes de valeurs)

```
Si expr = une des valeurs de la liste1 Alors
# instructions lorsque la valeur est dans liste1
Sinon Si expr = une des valeurs de la liste2 Alors
  # instructions lorsque la valeur est dans liste2
  Sinon Si ...
    Sinon Si expr = une des valeurs de la listeN Alors
    # instructions lorsque la valeur est dans listeN
    Sinon
    # instructions lorsque la valeur de la variable
    # ne se trouve dans aucune des listes précédentes
FinSi
```

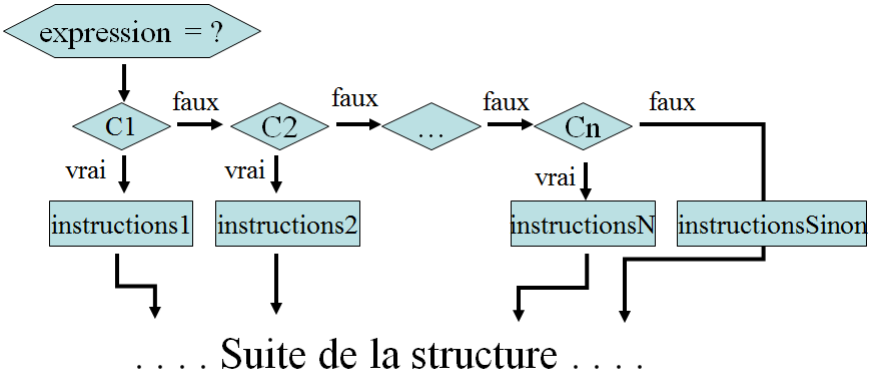
Sélective Selon (conditions)

```
Si condition1 Alors
# instructions lorsque la condition1 est vraie
Sinon Si condition2 Alors
  # instructions lorsque la condition2 est vraie
  Sinon Si ...
    Sinon Si conditionN Alors
    # instructions lorsque la conditionN est vraie
    Sinon
    # instructions à exécuter quand aucune
    # des conditions précédentes n'est vérifiée
FinSi
```

4.2 Sélective Selon (listes de valeurs)

Sélective Selon (listes de valeurs)

Elle évalue l'`expression` et n'exécute que les `instructionsI` qui correspondent à la valeur ordinale `Ci` (c.-à-d. de type entier ou caractère). La clause `Cas Autre` est facultative et permet de traiter tous les cas non traités précédemment. Il s'agit de l'instruction multi-conditionnelle classique des langages.



C/C++ **Sélective Selon (listes de valeurs)**

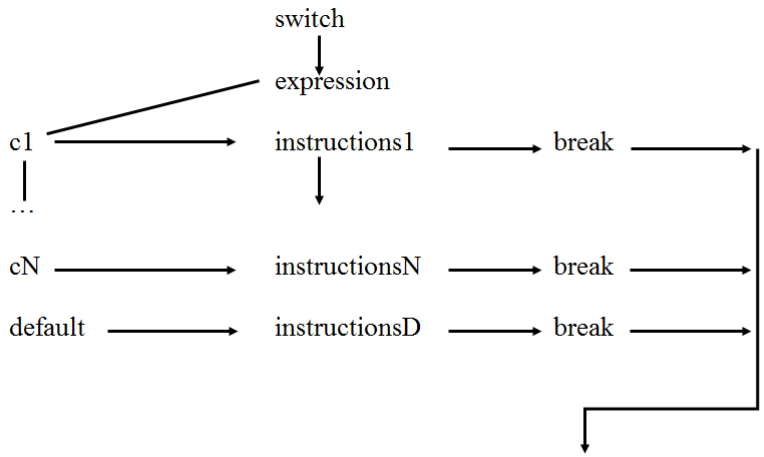
```
switch(expression)
{
  case C1:
    instructions1;
    break;
  ...
  case Cn:
    instructionsN;
    break;
  default:
    instructionsD
}
```

Remarque

Veillez à ne pas faire apparaitre une même valeur dans plusieurs listes.

C/C++ : Rupture

L'achèvement d'un énoncé n'est pas automatique : il faut l'expliciter à l'aide de l'instruction `break`.



Selon v.s. Si

Le `Selon` est moins général que le `Si` :

- L'expression doit être une valeur discrète (`Entier` ou `Caractère`).
- Les cas doivent être des *constantes* (pas de variables).

Si ces règles sont vérifiées, le `Selon` est plus efficace qu'une série de `Si` en cascade (car l'expression du `Selon` n'est évaluée qu'une seule fois et non en chacun des `Si`).

4.3 Exemple : Jour de la semaine en clair

L'algorithme saisit un jour de la semaine sous forme d'un nombre entier (0 pour dimanche, 1 pour lundi...) et affiche en clair le jour de la semaine pour un jour travaillé et « Week-end » pour le samedi ou le dimanche. Dans tous les autres cas, il affiche « Numéro de jour non valide ».

Programme @[pgjsem1.cpp]

```cpp
#include <iostream>
using namespace std;

int main()
{
  int jr;
  cout<<"Numero du jour? ";
  cin>>jr;
  switch (jr)
  {
    case 1:
      cout<<"lundi"<<endl;
      break;
    case 2:
      cout<<"mardi"<<endl;
      break;
    case 3:
      cout<<"mercredi"<<endl;
      break;
    case 4:
      cout<<"jeudi"<<endl;
      break;
    case 5:
      cout<<"vendredi"<<endl;
      break;
    case 0: case 6:
      cout<<"Week-end"<<endl;
      break;
    default:
      cout<<"Numero de jour non valide"<<endl;
  }
}
```

4.4 Opérateur Si-expression

Opérateur Si-expression

```
exprBool ? exprAlors : exprSinon
```

Explication

Évalue l'expression logique `exprBool` et si elle est vérifiée, effectue l'expression `exprAlors`, sinon l'expression `exprSinon`. Les `exprAlors` et `exprSinon` doivent être du même type.

Remarque

Cette syntaxe très raccourcie doit être réservée à de petits tests.

5 Spécificités C/C++

5.1 Ordre d'évaluation des opérandes

C/C++ : Ordre d'évaluation des opérandes
Elle n'est pas spécifié, excepté pour les opérateurs `&&` et `||`. Il faut en tenir compte lorsque l'un des opérandes est dépendant de l'autre, ce qui se produire en utilisant des opérateurs à effet de bord tels que `=`, `++` ou `+=`.

6 Références générales

Voici quelques liens concernant l'algèbre de Boole.

- Historique : George Boole
  http://fr.wikipedia.org/wiki/George_Boole
- Algèbre de Boole :
  http://fr.wikiversity.org/wiki/Logique_de_base/Alg%C3%A8bre_de_Boole