5

Conditions, opérateurs logiques et type booléen

Conditions simples

Dans la section précédente, nous avons introduit l'instruction de branchement conditionnel `'if'`.

```
if (condition) { ... }
else { ... }
```

Exemple

```
int main(){
       int n;
       cout << "Entrer un nombre entier: "; cin >> n;

       if (n > 10) { cout << "Nombre superieur a 10." << endl; }
       else { cout << "Nombre inferieur ou egal 10." << endl; }
}
```

Les conditions que nous avons employées jusqu'ici étaient des conditions simples qui consistaient par exemple à :

- Comparer une variable avec une valeur :

```
if  (n > 10) { ... }
else { ... }
```

- Comparer deux variables:

```
if  (a > b) { ... }
else { ... }
```

- Et en général, comparer deux expressions :

```
if  (a > 2 * b) { ... }
else { ...   }
```

ou

```
if  (2 * a < 3 * b + 1) { ... }
else { ...   }
```

Opérateurs de comparaison

Une condition simple compare deux expressions en utilisant les opérateurs de comparaison suivants :

```
<      inférieur à
>      supérieur à
<=     inférieur ou égal à
>=     supérieur ou égal à
==     égal à
!=     différent de
```

Attention L'opérateur pour tester si deux valeurs sont égales s'écrit avec deux signes égal (==). un seul signe égal (=) représente l'affectation. Par exemple, si on veut tester si une variable `a` est égale à `3`, il faut écrire:

`if (a == 3){ ... }` et non pas `if (a = 3){ ... }`

Cette dernière écriture est correcte et considérée comme licite par le compilateur mais elle est déconseillée. Référez-vous à la section "*6 - Erreurs commises par les débutants*" pour comprendre pourquoi.

Exercice : Écrire un programme C++ qui détermine si un nombre entier saisi au clavier est pair ou impair.

Astuce **:** On utilise l'opérateur modulo (%) qui renvoie le reste de la division entière.

`E.g., 11 % 4 = 3`

```
11 | 4
 3 | 2
```

```
int main(){
       int a;
       cout << "Entrer un nombre? "; cin >> a;

       if (a % 2) {
              cout << "a = " << a << endl;
              cout << "a est impair." << endl;
       }
       else {
              cout << "a = " << a << endl;
              cout << "a est pair." << endl;
       }
}
```

L'instruction : `if (a % 2) { ... } else { ... }`
est équivalente à : `if ((a % 2) != 0) { ... } else { ... }`

`(a % 2) != 0` : signifie que le reste de la division de `a` par `2` est non nul.

Remarquez bien que l'opérateur de comparaison "différent" s'écrit `!=` et non pas en utilisant le symbole ≠.

Conditions complexes

Nous venons de voir comment exprimer des conditions simples en utilisant des opérateurs de comparaison. En pratique, il est souvent nécessaire de combiner plusieurs de ces conditions simples. Voici un exemple :

Supposons qu'on vous demande d'écrire un programme qui permet de vérifier si un nombre entier est compris entre 5 et 8.

$$5 \leq n \leq 8$$

Comment vous allez faire ?

En C++, il est possible d'exprimer des conditions complexes (relier plusieurs conditions simples) en utilisant les opérateurs logiques : `and, or, not`. Donc, pour vérifier si un nombre entier `n` est compris entre 5 et 8, on écrit :

```
if ((n >= 5) and (n <= 8)) { ... }
else { ... }
```

Voyons les opérateurs logiques de près :

1. **Opérateur logique** `AND (&&)`

Exemple

```
if ((a < b) and (a < c)){ ... }
else { ... }
```

Cette condition est vraie uniquement si les deux conditions simples (`a < b`) et (`a < c`) sont toutes les deux vraies.

Condition 1	Condition 2	and
False	False	False
True	False	False
False	True	False
True	True	True

L'opérateur **`and`** peut aussi s'écrire `&&`

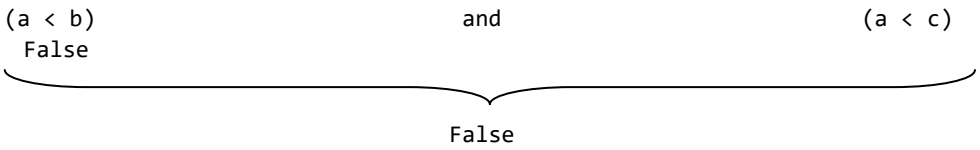
```
if ((a < b) && (a < c)){ ... }
else { ... }
```

Exemple 1

```
int a = 3;
int b = 2;
int c = 6;

if ((a < b) and (a < c)){ ... }
else { ... }
```

`a = 3 et b = 2`, donc la première condition simple `(a < b)` n'est pas vérifiée. Dans ce cas, le compilateur ne va pas passer et vérifier si la deuxième condition simple `(a < c)` est vraie. En fait, il va considérer que la condition complexe `(a < b) and (a < c)` comme fausse.



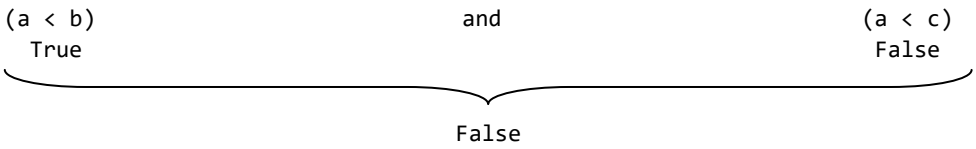
Exemple 2

```
int a = 3;
int b = 5;
int c = 2;

if ((a < b) and (a < c)){ ... }
else { ... }
```

`a = 3 et b = 5`, donc la première condition simple `(a < b)` est vérifiée. Dans ce cas, le compilateur va passer et vérifier si la deuxième condition simple `(a < c)` est vraie.

`a = 3 et c = 2`, donc la deuxième condition simple `(a < c)` n'est pas vérifiée. Alors, le compilateur va considérer que la condition complexe `(a < b) and (a < c)` comme fausse.



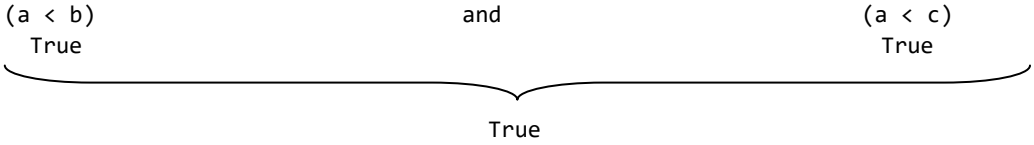
Exemple 3

```
int a = 3;
int b = 5;
int c = 6;

if ((a < b) and (a < c)){  ... }
else { ... }
```

`a = 3 et b = 5`, donc la première condition simple `(a < b)` est vérifiée. Dans ce cas, le compilateur va pas passer et vérifier si la deuxième condition simple `(a < c)` est vraie.

`a = 3 et c = 6`, donc la deuxième condition simple `(a < c)` est vérifiée. Alors, le compilateur va considérer que la condition complexe `(a < b) and (a < c)` comme vraie.



2. **Opérateur logique** `OR (||)`

Exemple

```
if ((a < b) or (a < c)){ ... }
else { ... }
```

Cette condition est vraie si au moins l'une des deux conditions `(a < b) ou (a < c )` est vraie.

Condition_1	Condition_2	Or
False	False	False
True	False	True
False	True	True
True	True	True

L'opérateur `or` peut aussi s'écrire `||`

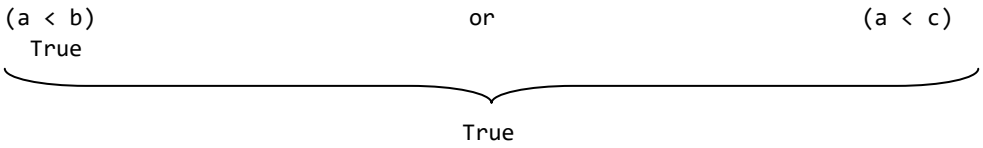
```
if ((a < b) || (a < c)){       ... }
else { ... }
```

Exemple 1

```
int a = 3;
int b = 5;
int c = 2;

if ((a < b) or (a < c)){   ... }
else { ... }
```

`a = 3 et b = 5`, donc la première condition simple `(a < b)` est vérifiée. Dans ce cas le compilateur ne va pas passer et vérifier si la deuxième condition simple `(a < c)` est vraie. En fait, le compilateur va considérer que la condition complexe `(a < b) or (a < c)` comme vraie.



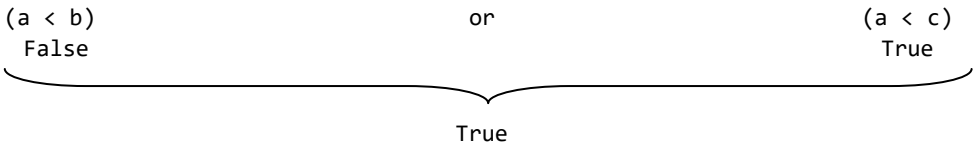
Exemple 2

```
int a = 3;
int b = 2;
int c = 6;

if ((a < b) or (a < c)){ ... }
else { ... }
```

`a = 3 et b = 2`, donc la première condition simple `(a < b)` n'est pas vérifiée. Dans ce cas, le compilateur va pas passer et vérifier si la deuxième condition simple `(a < c)` est vraie.

`a = 3 et c = 6`, donc la deuxième condition simple `(a < c)` est vérifiée. Dans ce cas, le compilateur va considérer que la condition complexe `(a < b) and (a < c)` comme vraie.



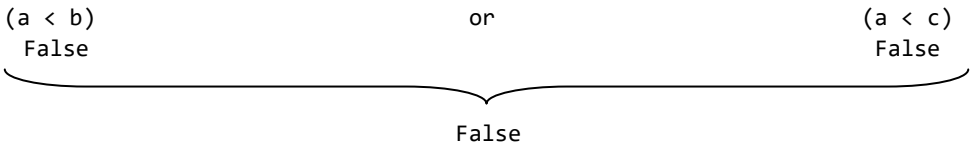
Exemple 3

```
int a = 3;
int b = 2;
int c = 1;

if ((a < b) and (a < c)){  ... }
else { ... }
```

`a = 3 et b = 2`, donc la première condition simple `(a < b)` n'est pas vérifiée. Dans ce cas le compilateur va pas passer et vérifier si la deuxième condition simple `(a < c)` est vraie.

`a = 3 et c = 1`, donc la deuxième condition simple `(a < c)` n'est pas vérifiée. Alors, le compilateur va considérer que la condition complexe `(a < b) and (a < c)` comme fausse.



3. Opérateur logique `Not (!)`

L'opérateur logique `not` est un opérateur unaire, il n'attend qu'un seul opérateur.

Exemple

```
if (not (a > b)){ ... }
else { ... }
```

Cette condition est vraie si `(a > b)` est fausse, et fausse si `(a > b)` est vraie.

Condition	not
False	True
True	False

L'opérateur **`not`** peut aussi s'écrire !

```
if (!(a > b)){ ... }
else { ... }
```

Exemple

```
int main(){
       int a, b;

       cout << "Entrer un nombre entier, a = ";   cin >> a;
       cout << "Entrer un deuxieme nombre, b = "; cin >> b;

       if (not (a > b)) { cout << "b est max." << endl; }
       else { cout << "a est max." << endl; }
}
```

Supposons que l'utilisateur a choisi `a = 3 et b = 5`, dans ce cas la condition `(a > b)` n'est pas vérifiée (elle retourne `false`). Ce qui veut dire que la condition `(not (a > b))` retournera `true` et le premier bloc sera exécuté :

```
cout << "b est max." << endl;
```

`a = 3 et b = 1`, dans ce cas la condition `(a > b)` est vérifiée (elle retourne `true`). Ce qui veut dire que la condition `(not (a > b))` retournera `false` et le deuxième bloc sera exécuté :

```
cout << "a est max." << endl;
```

`a = 3 et b = 3`, dans ce cas la condition `(a > b)` n'est vérifiée (elle retourne `false`). Ce qui veut dire que la condition `(not (a > b))` va retourner `true` et le premier bloc sera exécuté :

```
cout << "b est max." << endl;
```

Nous verrons des exemples d'utilisation de l'opérateur `not` plus loin dans la suite du cours.

Remarque : On peut relier plusieurs conditions simples et non pas seulement deux, en utilisant les différents opérateurs logiques. Par exemple :

```
1/     if ((a > b) and (a > c) or (b > c)){ ... }
       else { ... }

2/     if ((a > b) and (a > c) and (a > d)){ ... }
       else { ... }
```

Type booléen (anglais : boolean)

Les types que nous avons vus jusqu'ici :

```
int a;
double x;
```

Vous connaissez peut être d'autres types comme:

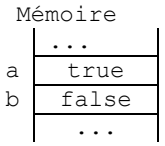
```
float y;
char c;
```

Dans ce qui suit nous allons parler du type `bool` (pour booléen):

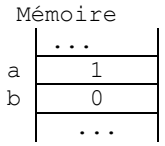
```
bool b;
```

Le type `bool` permet de déclarer une variable qui ne peut contenir que deux valeurs : `true` ou `false`. Donc, on peut initialiser des booléens à l'aide des constantes : `true` et `false`.

```
bool a = true;
bool b = false;
```



Comment l'ordinateur stocke ces deux valeurs (`true` et `false`) dans sa mémoire? En réalité, toute valeur nulle (0) est considérée comme `false` et toute autre valeur positive ou négative est considérée comme `true`.



Exemple 1 Écrire un programme C++ qui détermine si un nombre entier saisi au clavier est pair ou impair.

Astuce : On utilise l'opérateur module (%) qui renvoie le reste de la division entière.

```
int main(){
	int a;
	cout << "Entrer un nombre? "; cin >> a;

	bool nombrePair = true;
	if (a % 2) { nombrePair = false; }

	cout << "a = " << a;
	if (nombrePair) { cout << ", a est pair." << endl; }
	else { cout << ", a est impair." << endl; }
}
```

Exemple 2

```
int a = 1;
int b = 2;
bool c = a == b;
bool d = a < b;
```

Mémoire

	...
a	1
b	2
c	false
d	true
	...

Exemple 3 : On peut utiliser des opérateurs logiques (`and`, `or` et `not`) entre les booléens.

```
int a = 1;
int b = 2;
bool d = a == b;
bool e = (d or (a < b));
```

Mémoire

	...
a	1
b	2
d	false
e	true
	...

Exemple 4 : On peut utiliser des booléens comme des conditions.

```
int main(){
	int a, b;
	cout << "a? "; cin >> a;
	cout << "b? "; cin >> b;

	bool c = a < b;
	if (c) { cout << "a est inferieur a b"; }
	else { cout << "a est superieur ou egal a b"; }
}
```

Exercice Écrire un programme qui permet de vérifier si une valeur entrée au clavier est comprise entre 0 et 20.

```
int main(){
	int n;
	cout << "Entrer un nombre compris entre 0 et 20: ";
	cin >> n;

	if ((n >= 0) and (n <= 20)) {
		cout << "Le nombre que vous avez introduit est correct." << endl;
	}
	else {
		cout << "Le nombre que vous avez introduit est incorrect." << endl;
	}
}
```

Exercice : Ecrire un programme qui calcule Y tel que :

$$Y = \begin{cases} x + 8 & si \ \ x \leq 10 \\ x^2 + x + 10 & si \ \ 10 < x < 100 \\ 0 & si \ \ x \geq 100 \end{cases}$$

```
int main(){
	double x, y;
	cout << "x = "; cin >> x;

	if (x <= 10) { y = x + 8; }
	else if (x < 100){ y = x * x + x + 10; }
	else { y = 0; }

	cout << "y = " << y << endl;
}
```

Exercice : Écrire un programme qui permet de saisir deux valeurs entières, ensuite, vérifier si au moins l'une des deux est positive.

```
int main(){
	int a, b;
	cout << "Entrer un nombre entier : ";	cin >> a;
	cout << "Entrer un 2eme nombre entier : "; cin >> b;

	if ((a >= 0) or (b >= 0)) {
		cout << "Au moins un nombre est positif." << endl;
	}
	else {
		cout << "Les deux nombres sont negatifs." << endl;
	}
}
```

Exercice Un étudiant passe trois examens. Il est déclaré admis si la moyenne des trois examens est au moins égale à 10 points et la plus basse note est au moins égale à 8 points. Sinon, il est refusé. Ecrire le programme C++ correspondant.

```
int main(){
	double note1, note2, note3;
	cout << "Entrer la premiere note  : "; cin >> note1;
	cout << "Entrer la deuxieme note  : "; cin >> note2;
	cout << "Entrer la troisieme note : "; cin >> note3;

	double moyenne = (note1 + note2 + note3) / 3;
	cout << "Moyenne = " << moyenne << endl;

	if ((moyenne >= 10) and (note1 >= 8) and (note2 >= 8) and (note3 >= 8)){
		cout << "Etudiant admis!" << endl;
	} else {
		cout << "Etudiant refuse!" << endl;
	}
}
```