Information, Calcul et Communication

Composante Pratique: Programmation C++

MOOC semaine 2: instruction conditionnelle

Force d'un type et conversion ; conversion automatique

Booléen : opérateurs de comparaison, conversions

Instruction conditionnelle / instruction contrôlée

Opérateurs logiques / évaluation paresseuse

Choisir entre if-else ou switch

Opérateur ternaire

Force d'un type et conversion ; conversion automatique

Les opérateurs arithmétiques et logiques qui travaillent avec deux opérandes doivent avoir **le même type pour ces 2 opérandes**

Comment est évalué 5. / 9 ?

Comme ***5 / 9*** *ou comme* ***5. / 9.*** *?*

Le type *faible* → type *fort.* *Critère = domaine couvert le plus grand*

`int` → `long` → `long long` → `float` → `double` → `long double`

De plus il y a ces conversions automatiques pour les opérateurs arithmétiques:

`short`, `char` et **`bool`** → **`int`**

Booléen : opérateurs de comparaison, conversions

Un booléen ne peut prendre que 2 valeurs: `true` ou `false`

not ! ++ --	* / %	+ -	< <= > >=	== !=	and &&	or ||	? :

On dispose **d'opérateurs de comparaison** et **d'opérateur logique** pour construire une expression conditionnelle

Conversion entre les types de données et le type booléen :

- Dans une expression logique:
  - Motif binaire nul (0 , 0.0, …) → booléen **`false`**
  - Tous les autres motifs binaires → booléen **`true`**

```
int p(0);
cin >> p;
if(p) …    ⇔ if(p != 0) …
```

- Dans une expression arithmétique:

  `false` → `0` **`true`** → `1`

Sem3_MOOC2_SpeakUp1

L'expression C++ (1/2 + 1/2) est équivalente au booléen:

A : true

B : false

Instruction conditionnelle / instruction contrôlée

```
if( condition )
    une seule instruction contrôlée ;
else // optionnel
    une seule instruction contrôlée ;
```

1) Utiliser un bloc **{…} pour contrôler plus d'une instruction**

- l'indentation est demandée pour la lisibilité du code MAIS elle ne suffit PAS !!

- Si on dispose d'espace, toujours utiliser un bloc même avec une seule instruction contrôlée

Sem3_MOOC2_SpeakUp2: what is produced by this piece of code ?

```
...
int n(0);
if( n == 0 )
  cout << "n est nul" << endl;
  n = n + 1 ;
else
  cout << "n n'est pas nul" << endl;
...
```

A. Compilation: message d'erreur
B. A l'exécution: **`n est nul`**
C. A l'exécution: **`n n'est pas nul`**
D. A l'exécution: affiche les 2 messages

Un bloc **{…}** est équivalent à une seule instruction contrôlée

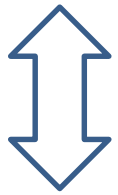
```
…
  int n(0);
  if( n == 0 )
  {
    cout << "n est nul";
    n = n + 1 ;
  }
  else
    cout << " n n'est pas nul";
 …
```

Une instruction '**else**' se rattache à la dernière instuction '**if**' qui n'a pas encore de '**else**'.

2) Ne PAS ajouter de point virgule en fin de ligne du if ou du else

le point virgule représente l'instruction nulle qui ne fait rien !!

```
int n(22);
if( n == 33 );
    cout << " n vaut 33 !" << endl;
```



```
int n(22);
if( n == 33 )
      ;   // l'instruction nulle est contrôlée !!

cout << " n vaut 33 !" << endl; // toujours exécutée
```

Instruction conditionnelle / instruction contrôlée (suite)

3) **Le piège classique**: utiliser l'opérateur **d'affectation =** au lieu de l'opérateur de **test d'égalité ==**

```
…
int n(22);
if( n = 0 )
      cout << " n est nul = " << n << endl;
else
      cout << " n n'est pas nul = " << n  << endl;
…
```

Sem3_MOOC2_SpeakUp3: what is produced by this piece of code ?

A. Compilation: message d'erreur / ne produit pas d'exécutable
B. A l'exécution, affichage de **`n est nul = 0`** et passe à la ligne
C. A l'exécution, affichage de : **`n est nul = 22`** et passe à la ligne
D. A l'exécution, affichage de : **`n'est pas nul = 22`** et passe à la ligne
E. A l'exécution, affichage de : **`n'est pas nul = 0`** et passe à la ligne
F. A l'exécution: affiche les 2 messages

Instruction conditionnelle / instruction contrôlée (suite)

Sem3_MOOC2_SpeakUp4: what is produced by this piece of code ?

```
…
float x(0.1);
if(x == 0.1) cout << "Egalité"     << endl;
else         cout << "Différence"  << endl;
…
```

A. Compilation: message d'erreur / ne produit pas d'exécutable
B. A l'exécution affiche **`Egalité`** et passe à la ligne
C. A l'exécution affiche **`Différence`** et passe à la ligne
D. A l'exécution: affiche les 2 messages
E. A l'exécution: aucun affichage

Opérateurs logiques / évaluation paresseuse

```
bool a, b;
```

a and b	b: false	b: true
a: false	false	false
a: true	false	true

ET-LOGIQUE

and &&

`false and b = false`

Evaluation paresseuse du ET logique:

L'opérande **droit** n'est PAS évalué si celui de **gauche** vaut **false**

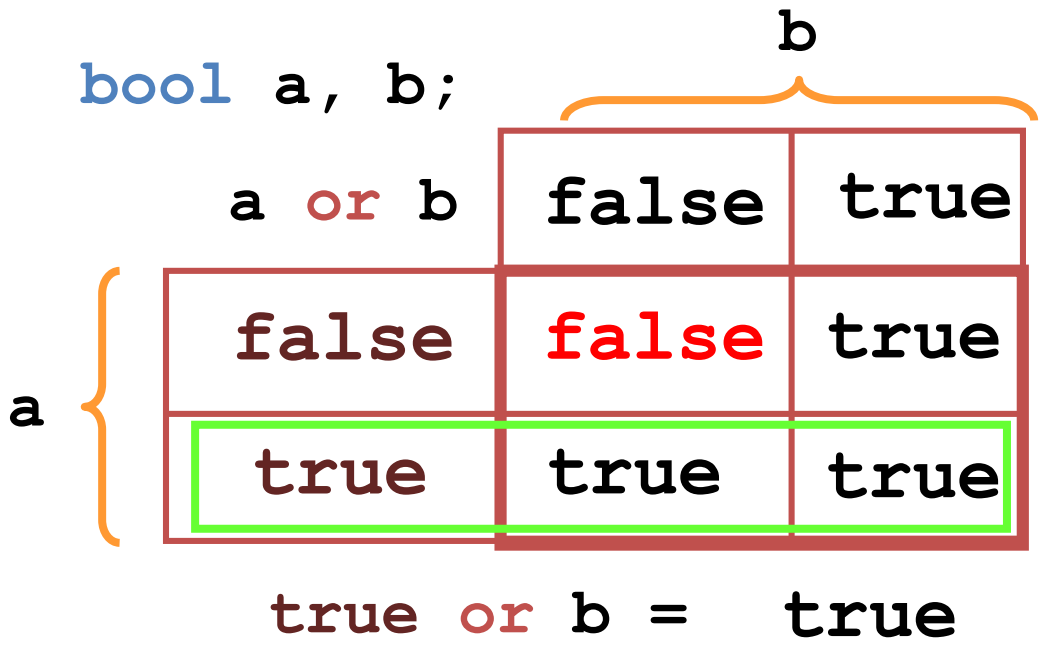
c'est inutile car on connaît déjà le résultat qui est `false`

Sem3_MOOC2_SpeakUp5

```
int n(0);
if((n!=0)and((100/n)==0))
    cout << "n>100" << endl;
```

A. Compilation: message d'erreur / pas d'exécutable
B. L'exécution n'affiche rien
C. L'exécution affiche `n>100` et passe à la ligne
D. L'exécution s'arrête car division par zéro

Opérateurs logiques / évaluation paresseuse (suite)



OU-LOGIQUE

or **||**

Evaluation paresseuse du OU logique:

L'opérande **droit** n'est PAS évalué si celui de **gauche** vaut **true**

c'est inutile car on connaît déjà le résultat qui est **`true`**

Sem3_MOOC2_SpeakUp6

```
int n(0);
if((n==0)or((100/n)==0))
    cout << "n>100 ou nul" << endl;
```

A. Compilation: message d'erreur / pas d'exécutable
B. L'exécution n'affiche rien
C. L'exécution affiche `n>100 ou nul` et passe à la ligne
D. L'exécution s'arrête car division par zéro

Négation des opérateurs logiques : lois de De Morgan

Exercice: exprimer **10 < n < 20** en C++

Sem3_MOOC2_SpeakUp7

```
int n(100);
if(10 < n < 20)
    cout << "10<n<20" << endl;
```

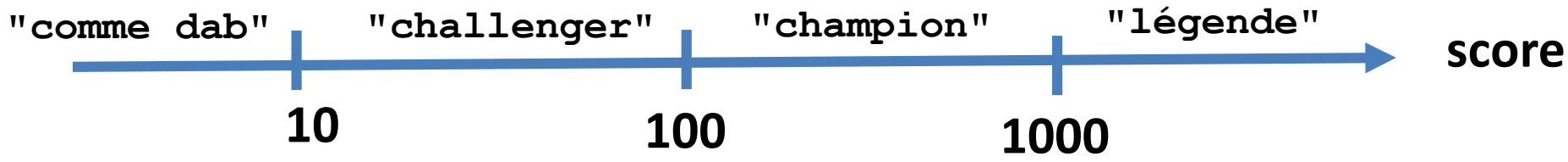
A. Compilation: message d'erreur / pas d'exécutable
B. L'exécution n'affiche rien
C. L'exécution affiche **10<n<20** et passe à la ligne

Exercice: exprimer la ***négation*** de **10 < n < 20**

Négation du **and** : `not(A and B) <-> (not A) or (not B)`

Négation du **or** : `not(A or B) <-> (not A) and (not B)`

Cascade de conditions : utiliser la syntaxe else if




```
…
  int score(0);
  cin >> score;
  if( score < 10 )
      cout << " comme dab   "   << endl;
  else if (score < 100)
      cout << " challenger "   << endl;
  else if( score < 1000)
      cout << " champion    "   << endl;
  else
      cout << " légende     "   << endl;
…
```

Switch sert à tester un ensemble de valeurs entières

```
1   int numero_mois(0);
2   cin >> numero_mois;
3
4   switch (numero_mois)
5   {
6     case 1:
7     case 3:
8     case 5:
9     case 7:
10    case 8:
11    case 10:
12    case 12: cout << "ce mois a 31 jours"          << endl; break;
13    case 4:
14    case 6:
15    case 9:
16    case 11: cout << "ce mois a 30 jours"          << endl; break;
17    case 2:  cout << "février a 28 ou 29 jours"    << endl; break;
19    default: cout << "Numéro de mois incorrect !« << endl; break;
20  }
```

Switch et incrémentation

Sem4_MOOC3_SpeakUp1: what is produced by this piece of code ?

A. **11** et passe à la ligne
B. **02** et passe à la ligne
C. **12** et passe à la ligne
D. **33** et passe à la ligne
E. aucune des autre réponses

```
…
int i(0);
switch(i++)
{
    case 0: cout << 0 ; break;
    case 1: cout << 1 ; break;
    case 2: cout << 2 ; break;
    default:cout << 3 ; break;
}
switch(++i)
{
    case 0: cout << 0 ; break;
    case 1: cout << 1 ; break;
    case 2: cout << 2 ; break;
    default:cout << 3 ; break;
}
cout << endl;
…
```

Evaluation d'une (sous-)expression contenant ++ ou --

La **pré-**incrémentation ou **pré-**décrémentation **modifie la valeur de la variable PUIS utilise sa nouvelle valeur** dans la suite pour l'évaluation de sous-expressions

```
int a(10),b(20),c(30), d(40);

a = ++b ;  // a prend la valeur :
c = --d ;  // c prend la valeur :
```

La **post-**incrémentation ou **post-**décrémentation utilise d'abord la valeur actuelle de la variable pour l'évaluation de sous-expression et **ensuite seulement modifie la valeur de la variable**

```
int a(10),b(20),c(30), d(40);

a = b++ ;  // a prend la valeur :
c = d-- ;  // c prend la valeur :
```

Opérateur conditionnel ternaire (*même priorité que l'affectation avec eval DG*)

not ! ++ --	* / %	+ -	< <= > >=	== !=	and &&	or ||	? :

```
condition ? opérande2 : opérande3
```

Si `condition` vaut **`true`**

Alors la valeur de l'expression est `opérande2`

Sinon la valeur de l'expression est `opérande3`

=> Écriture compacte, utile pour des expressions simples

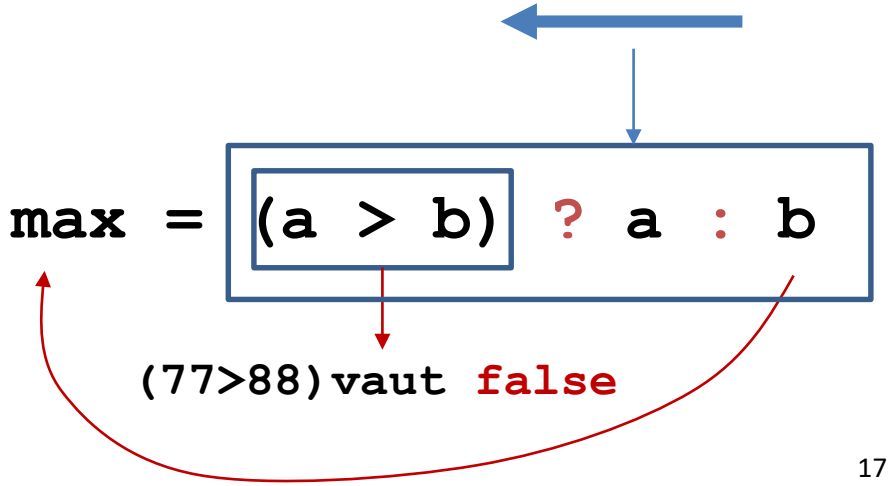
```
int max(0), a(77), b(88);

max = (a > b)?a:b ;

cout << max << endl;
```

Même basse priorité que l'affectation **=**

Mais associativité Droite-Gauche



Opérateur ternaire

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

int main()
{
      int a(222),b(33),c(44);
      cout << (a<b ? (a<c? a:c) : (b<c? b:c)) << endl;
      return 0;
}
```

Sem4_MOOC3_SpeakUp2: what is produced by this piece of code ?

A. Erreur de compilation
B. **222** et passe à la ligne
C. **33** et passe à la ligne
D. **44** et passe à la ligne
E. aucune des autre réponses