Information, Calcul, Communication (partie programmation) : Structures de contrôle en C++ (1) : branchements conditionnels

Jean-Cédric Chappelier

Laboratoire d'Intelligence Artificielle
Faculté I&C

Objectifs de la leçon d'aujourd'hui

- Ce qu'il faut savoir sur les premières **structures de contrôle** en C++
  - branchements
  - conditions
  - type `bool`
- Etude(s) de cas
- Réponses aux questions

Jean-Cédric Chappelier
& Jamila Sam
EPFL

Rappel du calendrier

		MOOC	décalage / MOOC	exercices prog. 1h45 Jeudi 8-10	cours prog. 45 min. Jeudi 10-11
1	11.09.25	--	-1	prise en main	Bienvenue/Introduction
2	18.09.25	1. variables	0	variables / expressions	variables / expressions
3	25.09.25	2. if	0	if – switch	if – switch
4	02.10.25	3. for/while	0	for / while	for / while
5	09.10.25	4. fonctions	0	fonctions (1)	fonctions (1)
6	16.10.25		1	fonctions (2)	fonctions (2)
-	23.10.25				
7	30.10.25	5. tableaux (vector)	1	vector	vector
8	06.11.25	6. string + struct	1	array / string	array / string
9	13.11.25		2	structures	structures
10	20.11.25	7. pointeurs	2	pointeurs	pointeurs
11	27.11.25		-	entrées/sorties	entrées/sorties
12	04.12.25		-	erreurs / exceptions	erreurs / exceptions
13	11.12.25		-	révisions	théorie : sécurité
14	18.12.25	8. étude de cas	-	révisions	Révisions
					(ne sont pas sur le MOOC)

Jean-Cédric Chappelier
& Jamila Sam

Les différentes structures de contrôle

On distingue 3 types de structures de contrôle :

les branchements conditionnels : *si ... alors ...*

C++

Si $\Delta = 0$
| $x \leftarrow -\frac{b}{2}$
Sinon
| $x \leftarrow \frac{-b-\sqrt{\Delta}}{2}$, $y \leftarrow \frac{-b+\sqrt{\Delta}}{2}$

les boucles conditionnelles : *tant que ...*

Tant que pas arrivé
| avancer d'un pas

Répéter
| poser la question
jusqu'à réponse valide

les itérations : *pour ... allant de ... à ...* , *pour ... parmi ...*

sem. prochaine

$$x = \sum_{i=1}^{5} \frac{1}{i^2}$$

$x \leftarrow 0$
Pour i de 1 à 5
| $x \leftarrow x + \frac{1}{i^2}$

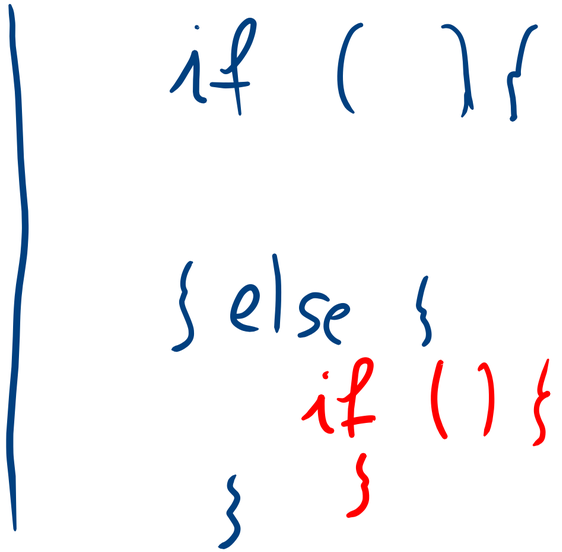
Jean-Cédric Chappelier
& Jamila Sam
EPFL

Branchement conditionnel

Le branchement conditionnel permet d'exécuter des traitements selon certaines conditions.

La syntaxe générale d'un branchement conditionnel est

```
if (condition)
    Instructions 1
else
    Instructions 2
```



La condition est tout d'abord évaluée puis, si le résultat de l'évaluation est vrai alors la séquence d'instructions 1 est exécutée, sinon la séquence d'instructions 2 est exécutée.

Instructions 1 et *Instructions 2* sont soit une **instruction élémentaire**, soit un **bloc d'instructions**.

Jean-Cédric Chappelier
& Jamila Sam
EPFL

Choix multiples

En C++, on peut écrire de façon plus synthétique l'enchaînement de plusieurs conditions dans le cas où l'on teste différentes valeurs d'une expression :

```
if (i == 1)
    Instructions 1
else if (i == 12)
    Instructions 2
...
else if (i == 36)
    Instructions N
else
    Instructions N+1
```

```
switch (i) {
  case 1:
    Instructions 1;
    break;

  case 12:
    Instructions 2;
    break;
    ...
  case 36:
    Instructions N;
    break;

  default:
    Instructions N+1;
    break;
}
```

4 conditions

valeurs ↑ entières connues

↳ sur expr.

Exemple plus complexe

Si on ne met pas de `break`, l'exécution ne passe pas à la fin du `switch`, mais continue l'exécution des instructions du `case` suivant :

2 ou 8

```
switch (a+b) {
  case 2:
  case 8: instruction2; // lorsque (a+b) vaut 2 ou 8
  case 4:
  case 3: instruction3; // lorsque (a+b) vaut 2, 3, 4 ou 8
    break;
  case 0: instruction1; // exécuté uniquement lorsque
    break;              // (a+b) vaut 0
  default: instruction4; // dans tous les autres cas
    break;
}
```


Jean-Cédric Chappelier
& Jamila Sam
EPFL

ATTENTION PIÈGE !

NON `if (x=3) {`

Ne pas confondre l'opérateur de test d'égalité `==` et l'opérateur d'affectation `=` !

`x = 3` : affecte la valeur 3 à la variable `x`
(et donc modifie cette dernière)

≡ `if (3)`
≡ `if (true)`

`x == 3` : teste la valeur de la variable `x`, renvoie `true` si elle vaut 3 et `false` sinon
(et donc ne modifie pas la valeur de `x`)

`if (3 == x)`

Évaluation « paresseuse »

Les opérateurs logiques `and` (ou `&&`) et `or` (ou `||`) effectuent une **évaluation « paresseuse »** (« *lazy evaluation* ») de leurs arguments :

l'évaluation des arguments se fait de la gauche vers la droite et seuls les arguments strictement nécessaires à la détermination de la valeur logique sont évalués.

Ainsi, dans `X1 and X2 and ... and Xn`, les arguments `Xi` ne sont évalués que *jusqu'au 1er argument faux* (s'il existe, auquel cas l'expression est fausse, sinon l'expression est vraie) ;

Exemple : dans `(x != 0.0) and (3.0/x > 12.0)` le second terme ne sera effectivement évalué uniquement si `x` est non nul. La division par `x` ne sera donc jamais erronée.

```
if (x != 0.0)
    if (3.0/x ...)
```

A et B et C

Jean-Cédric Chappelier
& Jamila Sam
EPFL

Évaluation « paresseuse »

Les opérateurs logiques `and` (ou `&&`) et `or` (ou `||`) effectuent une **évaluation « paresseuse »** (« *lazy evaluation* ») de leurs arguments :

l'évaluation des arguments se fait de la gauche vers la droite et seuls les arguments strictement nécessaires à la détermination de la valeur logique sont évalués.

Et dans `X1 or X2 or ... or Xn`, les arguments ne sont évalués que *jusqu'au 1er argument vrai* (s'il existe, auquel cas l'expression est vraie, sinon l'expression est fausse).

Exemple : dans `(x == 0.0) or (3.0/x <= 12.0)` le second terme ne sera effectivement évalué uniquement si `x` est non nul.

Jean-Cédric Chappelier
& Jamila Sam

Opérateurs

Opérateurs arithmétiques

`*`	multiplication	
`/`	division	
`%`	modulo	
`+`	addition	
`-`	soustraction	
`++`	incrément	*(1 opérande)*
`--`	décrément	*(1 opérande)*

Opérateurs de comparaison

`==`	teste l'égalité logique
`!=`	non égalité
`<`	inférieur
`>`	supérieur
`<=`	inférieur ou égal
`>=`	supérieur ou égal

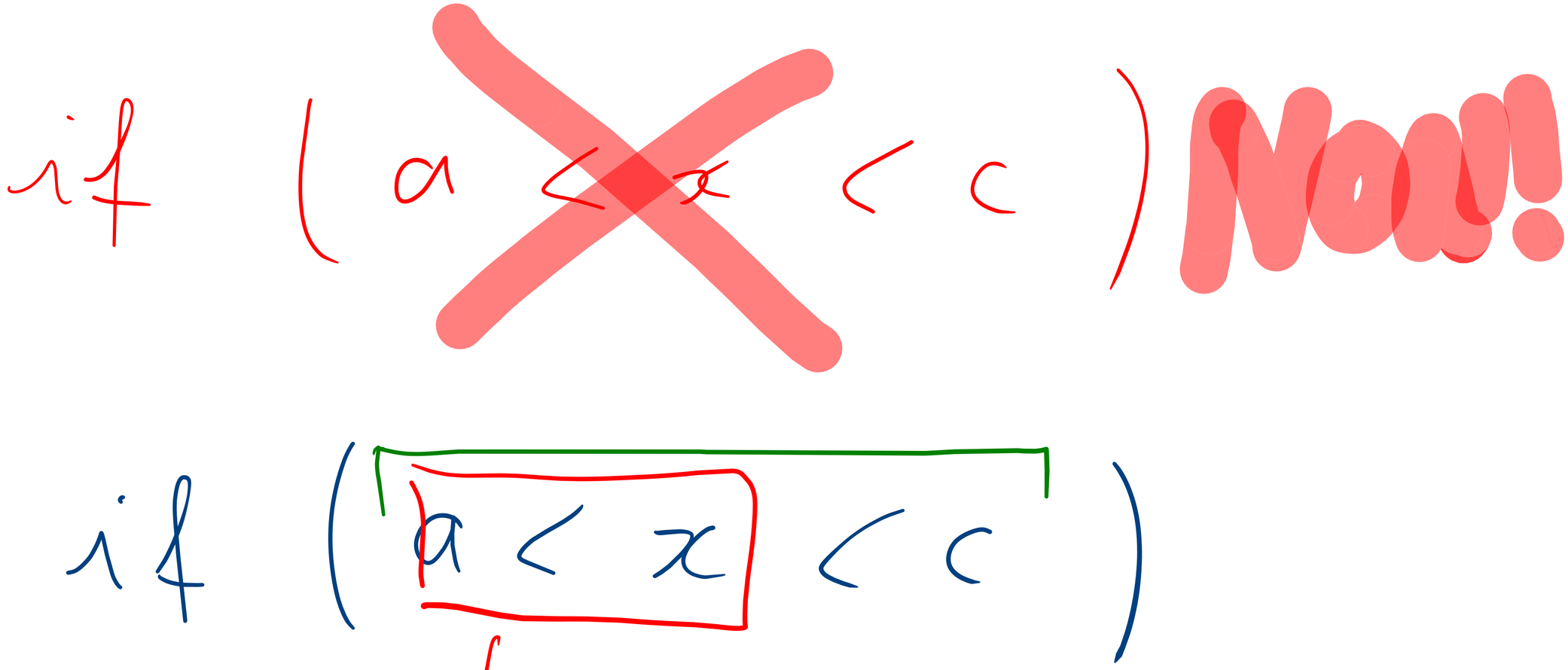
Opérateurs logiques

`and`	`&&`	« et » logique
`or`	`||`	ou
`not`	`!`	négation *(1 opérande)*

Priorités (par ordre décroissant, tous les opérateurs d'un même groupe sont de priorité égale) :

`not ++ --`, `* / %`, `+ -`, `< <= > >=`, `== !=`, `and`, `or`

Jean-Cédric Chappelier
& Jamila Sam
EPFL


if (a < x < c)
Nay!
if (a < x < c)
true → 1 < c
false → 0 < c

Etude(s) de cas

- reprendre l'équation du second degré
  - ☞ cf « exercice 0 »

- calculer (sans produire d'erreur) des valeurs de la fonction

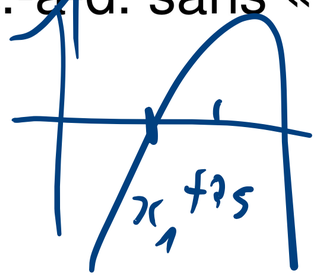
$$f(x) = \frac{\sqrt{20 + 7x - x^2}\,\log\left(\frac{1}{x+5}\right)}{\frac{x}{10} - \sqrt{\log\left(x^3 - 3x + 7\right) - \frac{x^2}{5}}}$$

Jean-Cédric Chappelier
& Jamila Sam
EPFL

Etude de cas

Comment calculer l'expression suivante sans produire d'erreur (c.-à-d. sans « `Nan` », « *Not a number* ») ?

$$\frac{\sqrt{20+7x-x^2}\,\log\left(\frac{1}{x+5}\right)}{\frac{x}{10}-\sqrt{\log\left(x^3-3x+7\right)-\frac{x^2}{5}}}$$



D_f if ((x > -5) and (x ≥ x1) and (x ≤ x2)

NON

Jean-Cédric Chappelier
& Jamila Sam

Etude de cas

Comment calculer l'expression suivante sans produire d'erreur (c.-à-d. sans « `Nan` », « *Not a number* ») ?

$$\frac{\sqrt{20+7x-x^2}\,\log\left(\frac{1}{x+5}\right)}{\frac{x}{10}-\sqrt{\log\left(x^3-3x+7\right)-\frac{x^2}{5}}}$$

☞ **DÉCOMPOSER**
Traiter « petit bout par petit bout »

☞ **VÉRIFIER** toute condition nécessaire **AVANT** utilisation

Par exemple :

```
if (abs(x + 5.0) < 1e-14) {
  cerr << "Expression invalide pour x=" << x
       << " : division par 0" << endl;
  return 1; // On sort avec un code d'erreur
}
```

Remarques (🐢 pour le moment) :

- pour les `double` : `abs(a - b) < petit` plutôt que `a == b`
  ☞ cf leçon I.4 de théorie (à venir)
- `cerr` est comme `cout` mais doit être préféré pour les messages d'erreur
- `return 1` : par convention : non 0 si erreur

Jean-Cédric Chappelier
& Jamila Sam

Etude de cas

Bien sûr, on suppose qu'au préalable `x` ait été déclaré et ait une valeur, par exemple saisie au clavier :

```
double x(0.0);
cout << "Entrez une valeur pour x : ";
cin >> x;
```

On pourrait alors continuer le code par exemple comme suit :

```
double x(0.0);
cout << "Entrez une valeur pour x : ";
cin >> x;

if (abs(x + 5.0) < 1e-14) {
  cerr << "Expression invalide pour x=" << x
       << " : division par 0" << endl;
  return 1;
}

if (x + 5.0 < 0.0) {
  cerr << "Expression invalide pour x=" << x
       << " : logarithme d'un nombre négatif" << endl;
  return 1;
}
```

Etude de cas

$$f(x) = \frac{\sqrt{20 + 7x - x^2}\,\log\left(\frac{1}{x+5}\right)}{\frac{x}{10} - \sqrt{\log\left(x^3 - 3x + 7\right) - \frac{x^2}{5}}}$$

Mais ce code présente un **gros défaut** !

```
if (abs(x + 5.0) < 1e-14) {
  cerr << "Expression invalide pour x=" << x
       << " : division par 0" << endl;
  return 1;
}

if (x + 5.0 < 0.0) {
  cerr << "Expression invalide pour x=" << x
       << " : logarithme d'un nombre négatif" << endl;
  return 1;
}
```

JAMAIS DE « COPIER-COLLER » !

Dans du code, il ne faut **jamais** avoir deux fois la même chose !
(= pour la même raison / de même origine [conceptuelle])

☞ problèmes de maintenance (corrections futures du code)

Jean-Cédric Chappelier
& Jamila Sam
EPFL

Etude de cas

Solution (générale) : introduire une variable auxiliaire,
qui réprésente justement le fait que ce soit la *même chose* (= la même expression *d'origine*) :

```
double auxiliaire(x + 5.0);
if (abs(auxiliaire) < 1e-14) {
  cerr << "Expression invalide pour x=" << x
       << " : division par 0" << endl;
  return 1;
}

if (auxiliaire < 0.0) {
  cerr << "Expression invalide pour x=" << x
       << " : logarithme d'un nombre négatif" << endl;
  return 1;
}
```

Note : dans ce cas précis, *particulier*, on pourrait bien sûr regrouper les deux tests dans un seul et même test unique,
mais comprenez bien le propos *général* !
(p.ex. si l'on *tient* à afficher un message d'erreur différent pour chacun des deux cas ci-dessus)

Etude de cas

$$f(x) = \frac{\sqrt{20+7x-x^2}\,\log\left(\frac{1}{x+5}\right)}{\frac{x}{10}-\sqrt{\log\left(x^3-3x+7\right)-\frac{x^2}{5}}}$$

On peut ensuite continuer dans le même esprit, en utilisant si nécessaire une seconde variable :

```
// ...
if (auxiliaire < 0.0) {
  cerr << "Expression invalide pour x=" << x
       << " : logarithme d'un nombre négatif" << endl;
  return 1;
}

double resultat(log(1.0 / auxiliaire));

auxiliaire = 20.0 + 7.0*x - x*x;
if (auxiliaire < 0.0) {
  cerr << "Expression invalide pour x=" << x
       << " : racine d'un nombre négatif" << endl;
  return 1;
}

resultat *= sqrt(auxiliaire);
// ...
```


Jean-Cédric Chappelier
& Jamila Sam
EPFL

```
int i(-5);
int j(2*i-1);

i = -4;

if (j == -9) {
  i = 2;
} else {
  i = 3;
}

if (j = 5) {
  i = 4;
} else {
  i = 5;
}

if (j == 0);
  i = 6;

cout << i << ", " << j << endl;
cout << j / i << endl;
```