Information, Calcul et Communication (partie programmation) : Branchements conditionnels

Jamila Sam

Laboratoire d'Intelligence Artificielle
Faculté I&C

Plan

- Quelques compléments de la semaine 2
- Approfondissements
  - Étude de cas
  - Du bon usage des booléens
  - Évaluation paresseuse
  - Choix multiples

Le langage C++ (1)

- Extension objet du langage C
- Développé initialement par Bjarn Stroustrup (1983-1985)
- Normalisé ISO en 1998, 2002 et 2011

Le C date de 1969-1973.

Le langage C++ (2)

Plus précisément, le langage C++ est un langage orienté-objet compilé fortement typé :

C++ = C + typage fort + objets

Parmi les avantages de C++, on peut citer :

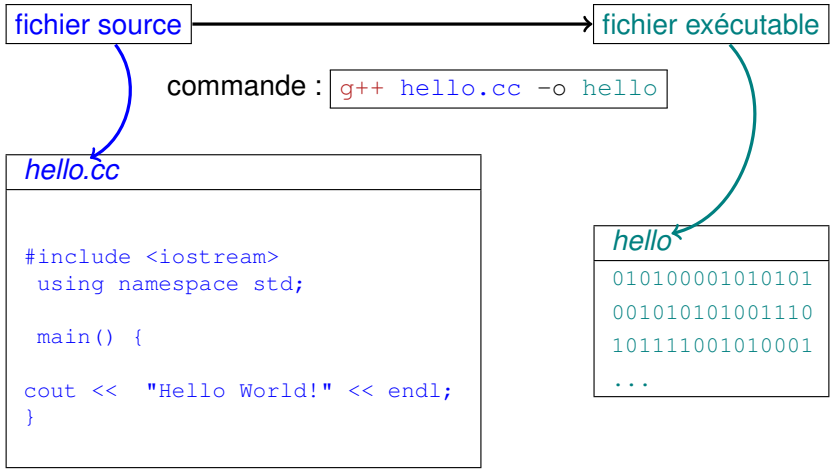
- Un des langages objets les plus utilisés
- Un langage compilé, ce qui permet la réalisation d'applications efficaces (disponibilité d'excellents compilateurs open-source (GNU))
- Un typage fort, ce qui permet au compilateur d'effectuer de nombreuses vérifications lors de la compilation $\Rightarrow$ moins de « bugs »...
- Un langage disponible sur pratiquement toutes les plate-formes ;
- Similarité syntaxique et facilité d'interfaçage avec le C

Le langage C++ (3)

. . . et les inconvénients :

- Similarité syntaxique avec le C !
- Pas nécessairement de gestion automatique de la mémoire
- Pas de protection de la mémoire
- Syntaxe parfois lourde et peu intuitive (“pousse-au-crime”)
- Gestion facultative des exceptions
- Effets parfois indésirables et peu intuitifs dus à la production automatique de code

Cycle de développement (1)



Cycle de développement (2)

Programmer c’est :

① réfléchir au problème ; **concevoir l’algorithme**
② traduire cette réflexion en un *texte* exprimé dans un langage donné (écriture du programme source)
③ traduire ce texte sous un format exécutable par un processeur (**compilation**, c++ ou g++)
④ exécution du programme

En pratique :

- erreurs de compilation (mal écrit)
- erreurs d’exécution (mal pensé)

$\Longrightarrow$ **correction(s)** ☞ d’où le(s) **cycle**(s) !

Données et traitements

Un programme C++ est donc à la base un ensemble de **traitements** s’effectuant sur des **données**.

Algorithme	structures de données
Traitements	**Données**
Expressions & Opérateurs Structures de contrôle Fonctions	Variables Portée Chaînes de caractères Tableaux statiques Tableaux dynamiques Structures Pointeurs
Entrées/Sorties	


auto

En C++11, on peut laisser le compilateur *deviner le type* d'une variable grace au mot-clé `auto`.

Le type de la variable est déduit du *contexte*. Il faut donc qu'il y ait un contexte, c'est-à-dire une initialisation.

Par exemple :

```
auto val(2);
auto j(2*i+5);
auto x(7.2835);
```

Conseil : **N'abuser pas** de cette possibilité et explicitez vos types autant que possibles.
N'utilisez `auto` que dans les cas « techniques », par exemple (qui viendra plus tard dans le cours) :

```
for (auto p = v.begin(); p != v.end(); ++p)
```

au lieu de

```
for (vector<int>::iterator
          p = v.begin(); p != v.end(); ++p)
```


Données modifiables/non modifiables

Par défaut, les variables en C++ sont modifiables.

Si l'on ne souhaite pas modifier une « variable » après son initialisation : la définir comme **constante** (pour ce nom là uniquement)

La nature **modifiable** ou **non modifiable** d'une donnée *au travers de ce nom* peut être définie lors de la déclaration par l'indication du mot réservé `const`.

Elle ne pourra plus être modifiée par le programme en utilisant ce nom (toute tentative de modification via ce nom produira un message d'erreur lors de la compilation).

Exemples :

```
int const couple(2);
double const g(9.81);
```



Expressions constantes

En C++11, il existe aussi le mot clé `constexpr`.

Il est d'utilisation plus générale, mais est aussi plus contraignant que `const` : la valeur initiale doit pouvoir être calculée à la compilation.

☞ Les deux (`const` et `constexpr`) sont donc **très différents** !

- `const` indique au compilateur qu'une donnée ne changera pas de valeur au travers de ce nom ; mais
  1. le compilateur peut très bien ne pas connaître la valeur en question au moment de la compilation ; et
  2. cette valeur pourrait changer par ailleurs.
- `constexpr` indique au compilateur qu'une donnée ne changera pas du tout de valeur et qu'il doit pouvoir en calculer la valeur au moment de la compilation (i.e. cette valeur ne dépend pas de ce qu'il va se passer plus tard dans le programme).

  Conseil : Si ces deux conditions sont vérifiées, on préfèrera utiliser `constexpr`.

Vidéos, transparents et quiz

`www.coursera.org/learn/initiation-programmation-cpp/`

☞ Semaine 2

J'écris à mes amis

```cpp
// Programme ami.cc
#include <iostream>
using namespace std;
int main()
{
        string nom;
        string adresse;

        // Lecture des donnee
        cout << "Donnez le nom de votre ami : " ;
        cin >> nom;

        cout << "Donnez l'adresse  de votre ami : " ;
        cin >> adresse;

    // Impression de l'etiquette

        cout << nom << endl;
        cout << adresse << endl;
}
```

☞ **Exécution linéaire**

EPFL

Structures de contrôle

C++ (comme la plupart des langages de programmation) permet la représentation d'enchaînements plus complexes grâce aux **structures de contrôle**

À quoi ça sert ?
Une structure de contrôle sert à **modifier l'ordre linéaire d'exécution** d'un programme.

☞ faire exécuter à la machine des tâches de façon *répétitive*, ou *en fonction de certaines conditions* (ou les deux).

EPFL

Les différentes structures de contrôle

On distingue 3 types de structures de contrôle :

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

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 réponse non valide
  poser la question

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

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

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

EPFL

Les différentes structures de contrôle

On distingue 3 types de structures de contrôle :

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

les boucles conditionnelles : *tant que ...*

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

Note : on peut toujours (évidemment !) faire des itérations en utilisant des boucles :

$x \leftarrow 0$
$i \leftarrow 1$
Tant que $i \leq 5$
  $x \leftarrow x + \frac{1}{i^2}$
  $i \leftarrow i + 1$

mais conceptuellement (et syntaxiquement aussi dans certains langages) il y a une différence.

EPFL

Les différentes structures de contrôle

On distingue 3 types de structures de contrôle :

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

les boucles conditionnelles : *tant que ...*

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

Les définitions de ces diverses structures de contrôle reposent sur les notions de **condition** et de **bloc** d'instructions.

Une **condition** est une *expression logique* telle que définie au cours précédent.

Retour à notre premier exemple

Résolution d'une équation du second degré : $x^2 + b\,x + c = 0$

```
#include <iostream>
#include <cmath>
using namespace std;
main() {
  double b(0.0);
  double c(0.0);
  double delta(0.0);

  cin >> b >> c;
  delta = b*b - 4*c;
  if (delta < 0.0) {
    cout << "pas de solutions reelles" << endl;
  } else if (delta == 0.0) {
    cout << "une solution unique : " << -b/2.0 << endl;
  } else {
    cout << "deux solutions : " << (-b-sqrt(delta))/2.0
         << " et " << (-b+sqrt(delta))/2.0 << endl;
  }
}
```

données
traitements
structures de contrôle

Conditions

Pour exprimer des conditions

☞ Opérateurs de **comparaison** et opérateurs **logiques**

Opérateurs de comparaison

Les opérateurs de comparaison (relationnels) sont :

`==`	égalité
`!=`	non égalité
`<`	inférieur
`>`	supérieur
`<=`	inférieur ou égal
`>=`	supérieur ou égal

Leur résultat est un **booléen** (`true` ou `false`)
Exemples (expressions logiques avec opérateur de comparaison) :

```
x >= y
x != (z + 2)
(x + 4) - z == 5
b = (x == 5);
```

Le type `bool`

`bool` est un type (au même titre que `char`, `int` ou `double`)

- ne peux prendre que **deux valeurs**
- valeurs littérales : `true`, `false`
- représente de « valeurs de vérité », des conditions logiques

Opérateurs logiques

On peut combiner des expressions logiques au moyen d'opérateurs logiques :

`&&`	"et" logique
`||`	ou
`!`	négation

*(Remarque : cet opérateur n'a qu'**un seul** opérande)*

Exemples :

- Expression logique utilisant des opérateurs logiques :

```
((z != 0) && (2*(x-y)/z < 3))
```

- Code utilisant des opérateurs logiques :

```
bool un_test(true);
bool un_autre_test((x >= 0) ||
                   ((x*y > 0) && !unTest));
```

Note : La norme (ISO/IEC 14882 :1998) définit aussi les formes alternatives : `and`, `or` et `not` Par exemple `((x >= 0) or ((x*y > 0) and not un_test))`

☞ pas toujours supporté par tous les compilateurs :-(

Opérateurs logiques (2)

Les opérateurs logiques `&&`, `||` et `!` sont définis par les tables de vérité usuelles :

x	y	!x	x && y	x || y	x ∧ y
true	true	false	true	true	false
true	false	false	false	true	true
false	true	true	false	true	true
false	false	true	false	false	false

Approfondissements

- Du bon usage des booléens
- Évaluation paresseuse
- Choix multiples

Du bon usage des variables booléennes

Une **variable booléenne** représente une **condition**

☞ Inutile de la comparer explicitement à `true` ou `false` !

Correct :

```
if (un_test)
if (!un_test)
return un_test;
```

Non recommandé :

```
if (un_test == true)
if (un_test != true)
if (un_test == false)
if (un_test != false)
```

Évaluation « paresseuse »

Les opérateurs logiques `&&` et `||` effectuent une **évaluation « paresseuse »** (“*lazy evaluation*”) de leur 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 && X2 && ... && 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 `(i != 0) && (3/i < 25)` le second terme ne sera effectivement évalué uniquement si `i` est non nul. La division par `i` ne sera donc jamais erronée.
Et dans `X1 || X2 || ... || 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 `(i == 0) || (3/i < 25)` le second terme ne sera effectivement évalué uniquement si `i` est non nul.

Choix multiples

On peut écrire de façon plus claire l'enchaînement de plusieurs conditions dans le cas où l'on teste différentes valeurs d'une expression

Avec `if ..else`

```
if (i == 1)
   Instructions1
else if (i == 12)
   Instructions2
else if ...
else
   InstructionsN+1
```

Avec `switch`

```
switch (i)
{
   case 1:
      Instructions1
      break;
   case 12:
      Instructions2
      break;
   case ...
   default:
      InstructionsN+1
}
```

☞ chaque `case` correspond à une constante `int` (ou équivalent) ou `char`

To `break` or not to `break` ...

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

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

`switch` : un exemple

Soit l'enchaînement de conditions suivant :

```
cout << "Entrez un entier: ";

int a; cin >> a;

if (a == 0)
    System.out.println("To break");
else
    if (a == 1)
        cout << "or not" << endl;
    else
        if (a == 2)
            cout << "to break" << endl;
        else
            cout << "that is the question" << endl;
```

Exercice : essayons de l'exprimer au moyen d'un `switch` ...

Avec `break`

Code

```
cout << "Entrez un entier: ";
int a; cin >> a;

switch (a) {
case 0 :
        cout << "To break" << endl;
  break;
case 1 :
        cout << "or not" << endl;
  break;
case 2 :
        cout << "to break" << endl;
  break;
default :
        cout <<
      "that is the question" << endl;
}
```

Exécution

```
Entrez un entier: 0
To break

Entrez un entier: 1
or not

Entrez un entier: 99
that is the question
```


Sans `break`

Code

```
cout << "Entrez un entier: ";
int a; cin >> a;

switch (a) {
case 0 :
        cout << "To break" << endl;
case 1 :
        cout << "or not" << endl;
case 2 :
        cout << "to break" << endl;
default :
        cout <<
      "that is the question" << endl;
}
```

Exécution

```
Entrez un entier: 99
that is the question

Entrez un entier: 2
to break
that is the question

Entrez un entier: 0
To break
or not
to break
that is the question
```


`switch` vs `if..else`

`switch` est moins général que **`if..else`** :

- La valeur sur laquell on teste doit être soit **`char`** ou **`int`**
- Les cas **doivent être des constantes** (pas de variables)

Etude de cas

- reprendre l'équation du second degré
  - ☞ cf « exercice 0 »
- calculer des valeurs de la fonction

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

Etude de cas

Comment calculer l'expression suivante sans produire d'erreur (i.e. 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 »

Par exemple :

```
if (x + 5.0 == 0.0) {
  cerr << "Expression invalide pour x=" << x
       << " : division par 0" << endl;
  return 1; // On sort avec un code d'erreur
}
```


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 (x + 5.0 == 0.0) {
  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

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

```
if (x + 5.0 == 0.0) {
  cerr << "Expression invalide pour x=" << x
       << " : division par 0" << endl;
  return 1;
}

if (x + 5.0 < 0.0) {  //  x + 5.0 COPIEE-COLLEE !!
  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 !

☞ problèmes de maintenance (corrections futures du code)

Etude de cas

Solution : introduire une variable auxiliaire,
qui réprésente justement le fait que ce soit la *même chose* :

```
double auxiliaire(x + 5.0);
if (auxiliaire == 0.0) {
  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;
}
```


Etude de cas

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);

// etc.
```


Qu'en est-il des problèmes de précision ?

Les comparaisons exactes entre double peuvent être erronées en raison de problèmes de précision.

Ainsi x+0.5 == 0 peut (alors que l'on pense x valoir -5.0) retourner false au lieu de true si par exemple x est obtenue par calcul !

En pratique, **lorsque cela est rendu possible par le domaine d'application**, on fait souvent des comparaisons à un *espilon près* :

```
constexpr PRECISION(1e-8);


if (abs(x + 0.5) <= PRECISION) {
        //...
}
```

Néamoins, il ne s'agit alors plus d'une résolution mathématique du problème !
Si l'on veut faire des mathématiques avec un ordinateur, il faut soit admettre que les résultats ne sont pas garantis **(ce que vous allez faire dans les exercices de cette semaine)**, soit avoir recours à d'autres procédés plus complexes (arithmétique des intervalles par exemple, mais c'est tout un domaine !)

Pour préparer le prochain cours

- Vidéos et quiz du MOOC semaine 3 :
  - Itérations : introduction [12 :37]
  - Itérations : approfondissement et exemples [19 :17]
  - Itérations : quiz [09 :04]
  - Boucles conditionnelles [22 :31]
  - Blocs d'instructions [12 :18]
- Le prochain cours :
  - résumé et quelques approfondissements