Information, Calcul et Communication (partie programmation) : Variables et Expressions

Jamila Sam

Laboratoire d'Intelligence Artificielle
Faculté I&C

EPFL

Vidéos, Quiz et transparents

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

☞ Semaine 1

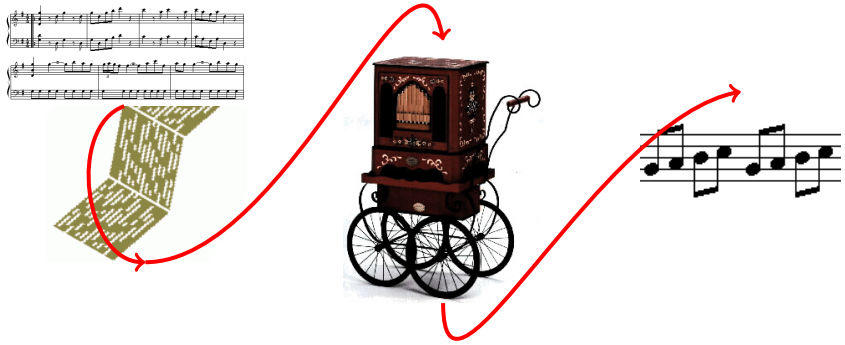
EPFL

Objectifs de la leçon d'aujourd'hui

- Résumer ce qu'il faut avoir retenu des premières leçons du MOOC :
  - développement d'un programme
  - variables
  - types
  - expressions
- Etude de cas (très simple ici)
- Compléments de cours :
  - un tout petit peu plus sur les types, dont `char`
  - `auto`
  - `const`/`constexpr`
- Répondre à vos questions

EPFL

Qu'est-ce que la programmation ?



«PROGRAMME» :

Conception : quelles notes enchainer ?

Réalisation : percer les trous aux bons endroits

Exécution : tourner la manivelle

Résultat : mélodie

EPFL

Algorithmes – objectifs

On attend d'un algorithme qu'il :

- se termine,
- produise un résultat correct,
- pour toute donnée d'entrée valable.

☞ Il n'y a (mal)heureusement aucune recette pour produire un algorithme !

Difficulté de l'Informatique

Un programme doit être valable pour toute une gamme d'entrées !

☞ Impossible de vérifier par des essais : on ne pourra jamais tester tous les cas.

Vérification par preuves mathématiques.

Importance du travail soigneux et mûrement réfléchi !

Algorithme $\neq$ Programme

Un algorithme est **indépendant du langage de programmation** dans lequel on va l'exprimer **et de l'ordinateur** utilisé pour l'exécuter

C'est une description abstraite des étapes conduisant à la solution d'un problème.

Algorithme = partie conceptuelle d'un **programme** (indépendante du langage)

Programme = implémentation (i.e., réalisation) de l'**algorithme**, dans un langage de programmation et sur un système particulier.

Les instructions de l'ordinateur

Concrètement, quelles sont les instructions et les données « adaptées » à l'ordinateur ?

Ordinateur $\simeq$

microprocesseur
: détermine l'ensemble des instructions élémentaires que l'ordinateur est capable d'exécuter ;

mémoire centrale
: détermine l'espace dans lequel des données peuvent être stockées en cours de traitement

périphériques
: permettent l'échange ou la sauvegarde à long terme des données

Les instructions de l'ordinateur

Concrètement, quelles sont les instructions et les données « adaptées » à l'ordinateur ?

Ordinateur ≃
microprocesseur
mémoire centrale
périphériques

☞ C'est donc **le microprocesseur** qui **détermine le « jeu d'instructions »** (et le type de données) à utiliser.

On les appelle « Instructions Machine », « Langage Machine », ≃ « Assembleur »

On peut programmer directement le microprocesseur en langage machine...
...mais c'est un peu fastidieux et de plus, chaque processeur utilise ses propres instructions

Exemple d'instructions-machine

Programme en Assembleur
`1: LOAD 10 5`
`2: CMP 10 0`
`3: JUMP +3`
`4: DECR 10`
`5: JUMP -3`
`6: END`

mettre 5 dans la mémoire 10
comparer le contenu de la mémoire 10 à 0
si tel est le cas sauter 3 instructions plus loin
décrémenter la mémoire 10 (de 1)
sauter 3 instructions en arrière

Instructions machine :

Instructions	Code Machine	données	Code Machine
CMP	00000000	-3	10000011
DECR	00000001	0	00000000
END	00000010	2	00000010
JUMP	00000011	3	00000011
LOAD	00000100	5	00000101
		6	00000110
		10	00001010

Le programme ci-dessus correspond donc physiquement en machine à la séquence :

```
000001000000101000000101000000000000101000000000000000011000000110000000100000101000000011100000110000010
```


La notion de langage de programmation

Cependant, ces instructions-machine sont **trop élémentaires** pour pouvoir être efficacement utilisées (par les humains) pour l'écriture de programmes...

. . . il faut donc fournir au programmeur la possibilité d'**utiliser des instructions de plus haut niveau**, plus proches de notre manière de penser et de conceptualiser les problèmes . . .

Exemples de langage de programmation de haut niveau :

« *vieux* » BASIC	C

`1 N=5`
`2 IF (N>0) THEN PRINT N;`
`  N=N-1; GOTO 2`
`3 END`
	`main() {`
	`int n;`
	`for (n=5; n>0; --n)`
	`  printf("%d\n", n);`
	`}`

La notion de langage de programmation (2)

Comment rendre les instructions plus sophistiquées compréhensibles par l'ordinateur ?

☞ **traduire** les séquences d'instructions de haut niveau en instructions-machine directement exécutables par le microprocesseur

Selon ses caractéristiques, un tel traducteur est appelé **compilateur** ou **interpréteur**

L'ensemble des instructions de plus haut niveau qu'un compilateur ou un interpréteur est capable de traiter constitue un **langage de programmation**.

La notion de langage de programmation (3)

Un **langage de programmation** est donc un moyen formel permettant de décrire des *traitements* (i.e. des tâches à réaliser) sous la forme de *programmes* (i.e. de séquences d'instructions et de données de « haut niveau », compréhensibles par le programmeur) pour lesquels un **compilateur** ou un **interpréteur** est disponible pour permettre l'exécution effective par un ordinateur.

Exemples de langages de programmation : C, python, Lisp, Java... et C++

Interpréteur/Compilateur

Comment rendre les instructions plus sophistiquées compréhensibles par l'ordinateur ?

☞ **traduire** les séquences d'instructions de haut niveau en instructions-machine directement exécutables par le microprocesseur

Selon ses caractéristiques, un tel traducteur est appelé **compilateur** ou **interpréteur**.

L'ensemble des instructions de plus haut niveau qu'un compilateur ou un interpréteur est capable de traiter constitue un **langage de programmation**.

Interpréteur/Compilateur (2)

- Le compilateur traduit un programme, écrit dans un langage de haut niveau, en un fichier binaire (exécutable) spécifique à une architecture matérielle (ARM, x86, amd64, ...). Chaque plateforme a son format de fichiers binaires (ELF, COFF, ...)
- L'interpréteur exécute un programme, écrit dans un langage de haut niveau, sans étape intermédiaire. Un programme interprété de manière naïve est plus lent qu'un programme compilé, mais indépendant de l'architecture.

☞ C++ est un langage compilé

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 :

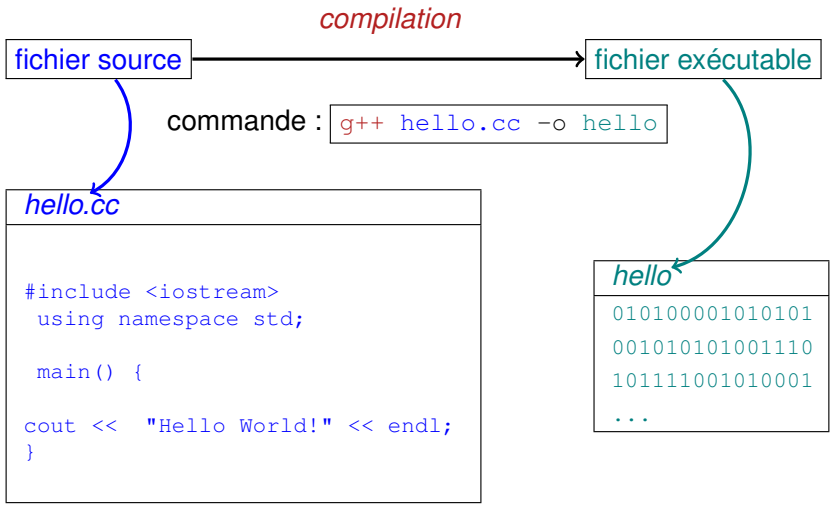
- Un des langages objets les plus utilisés
- Un langage compilé, ce qui permet la réalisation d'applications efficaces (disponibilité d'excellents compilateurs open-source (GNU))
- Un typage fort, ce qui permet au compilateur d'effectuer de nombreuses vérifications lors de la compilation ⇒ moins de « 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)

⟹ **correction(s)** ☞ d'où le(s) **cycle**(s) !

Et l'IA dans tout cela ...

Les outils à base d'IA générative (ChatGPT, Github-Copilot, Claude etc.) révolutionnent désormais l'approche au codage

Ils sont déjà considérés, à juste titre, comme des supports indispensables

Attention ! **Ils ne sont pas fait pour l'apprentissage.**

Analogie : utiliser une machine à calculer sans connaître les bases de l'arithmétique !

☞ Impossible de comprendre pourquoi un calcul est faux par exemple !

Et l'IA dans tout cela ...

Derrière l'écriture d'un programme, il y a des enjeux de **modélisation** !

Si l'on ne comprends ce qui caractérise une bonne modélisation on ne saura pas interroger une IA de façon adéquate

☞ Dans «la vraie vie» les spécifications des programmes ne sont pas écrites pour vous !

Et l'IA dans tout cela ...

Pour ce cours, du point de vue de l'enseignement, nous allons faire comme si les outils d'IA n'existaient pas

☞ **Le but n'est pas de vous apprendre à interroger ces outils mais de vous enseigner ce qui constitue un (bon) programme, techniquement et méthodologiquement.**

De votre côté, l'usage de ces outils est toléré pour des tâches simples et comme support à la compréhension

☞ Il est indispensable de vous **re-approprier** le matériel ou les explications produites par ces outils en vous montrant capable de les comprendre, de les re-expliquer par vous même et d'y porter un regard critique.

Attention ! Des questions d'examen porteront sur le mini-projet !

Structure générale d'un programme C++

La structure très générale d'un programme C++ est la suivante :

```
#include <des_librairies_utiles>
...

using namespace std; // on y reviendra

(déclaration d'objets globaux)                [à éviter]

déclarations de fonctions utiles             [recommandé]

int main()  //ou int main(int argc, char **argv)
{
corps du
programme principal              [si possible assez court]
}
```

Exemple de programme en C++

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

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**
	Variables
Expressions & Opérateurs	
Structures de contrôle	
Fonctions	Portée
	Chaînes de caractères
	Tableaux statiques
	Tableaux dynamiques
	Structures
	Pointeurs
Entrées/Sorties	

Variables

En C++, une **valeur** à conserver est stockée dans une variable caractérisée par :

- son **type**
- et son **identificateur** ;

(définis lors de la **déclaration**)

La *valeur* peut être définie une première fois lors de l'**initialisation**, puis éventuellement modifiée par la suite.

Rappels de syntaxe :

`type nom ;`	(déclaration)
`type nom(valeur);`	(initialisation)
`nom = expression ;`	(affectation)

Types élémentaires :

```
int
double
char
bool
```

Exemples :

```
int val(2) ;
const double z(x+2.0*y);
constexpr double pi(3.141592653);
i = j + 3;
```


Variables et types

A retenir :

- variable = représentation interne d'une « donnée » du problème traité = une *valeur*
- en C++, les valeurs (donc aussi les variables) sont **typées** : « nature »/« ensemble d'appartenance » de la valeur
- Les principaux **types élémentaires** définis en C++ sont :
  - **`int`** : (une partie des) nombres entiers
  - **`double`** : (une partie des) nombres décimaux
  - **`bool`** : les valeurs logiques « *vrai* » (`true`) et « *faux* » (`false`)
  - **`char`** : les caractères (`'a'`, `'!'`, ...)

Note : nous verrons plus tard d'autres types :
les types composés, les types énumérés et les types synonymes (alias de typeS).

Valeurs Littérales

- valeurs littérales de type `int` : `1, 12, ...`
- valeurs littérales de type `double` : `1.23, ...`
  Remarque :
  `12.3e4` correspond à $12.3 \cdot 10^4$ (soit `123000`)
  `12.3e-4` correspond à $12.3 \cdot 10^{-4}$ (soit `0.00123`)
- valeurs littérales de type `char` : `'a', '!', ...`
  Remarque :
  le caractère `'` se représente par `\'`
  le caractère `\` se représente par `\\`
- valeurs littérales de type booléen : `true, false`

Remarque : la valeur littérale `0` est une valeur d'initialisation qui peut être affectée à une variable de n'importe quel type.

Opérateurs

Operateurs arithmétiques

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

Operateurs de comparaison

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

Operateurs logiques

`&&`	"et" logique	
`||`	ou	
`^`	ou exclusif	
`!`	négation	*(1 opérande)*

Priorités (par ordre décroissant, tous les opérateurs d'un même groupe sont de priorité égale) :
`!` `++` `--`, `*` `/` `%`, `+` `-`, `<` `<=` `>` `>=`, `==` `!=`, `^` `&&`, `||`

Expression

- une expression représente un calcul à faire, à évaluer
- expression = combinaison d'expressions, de valeurs, de variables, à l'aide d'opérateurs
- toute expression a un type (et une valeur) :
  en C++, **toute expression *fait* quelque chose et *vaut* quelque chose**

« Etude de cas »

Qu'affiche le code suivant :

(à découvrir en classe)


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`.

Pour préparer le prochain cours

- Vidéos et quiz du MOOC semaine 2 :
  - Branchements conditionnels [13 :46]
  - Conditions [13 :14]
  - Erreurs de débutant le type bool [14 :34]
- Le prochain cours :
  - de 12h15 à 13h (résumé et quelques approfondissements)