Objectifs
Apprendre
Résumé
Etude de cas
Compléments

Information, Calcul, Communication (partie programmation) : VARIABLES & EXPRESSIONS

Jean-Cédric Chappelier

Faculté I&C

Rappel du calendrier

		MOOC	décalage / MOOC	exercices prog. 1h45 Jeudi 8-10	cours prog. 45 min. Jeudi 10-11
1	10.09.26	--	-1	prise en main	Bienvenue/Introduction
2	17.09.26	1. variables	0	variables / expressions	variables / expressions
3	24.09.26	2. if	0	if – switch	if – switch
4	01.10.26	3. for/while	0	for / while	for / while
5	08.10.26	4. fonctions	0	fonctions (1)	fonctions (1)
6	15.10.26		1	fonctions (2)	fonctions (2)
-	22.10.26				
7	29.10.26	5. tableaux (vector)	1	vector	vector
8	05.11.26	6. string + struct	1	array / string	array / string
9	12.11.26		2	structures	structures
10	19.11.26	7. pointeurs	2	pointeurs	pointeurs
11	26.11.26		-	entrées/sorties	entrées/sorties
12	03.12.26		-	erreurs / exceptions	erreurs / exceptions
13	10.12.26		-	révisions	théorie : sécurité
14	17.12.26	8. étude de cas	-	révisions	Révisions

Objectifs de la leçon d'aujourd'hui

- Apprendre à programmer / Rôle de l'IA
- Résumer ce qu'il faut avoir retenu des premières leçons :
  - variables
  - types
  - expressions
- Etude de cas (très simple ici)
- Compléments de cours :
  - `auto`
  - `const`/`constexpr`
- Répondre à vos questions

Jean-Cédric Chappelier
& Jamila Sam

Bien apprendre / Rôle de l'IA

Qu'est-ce qui est censé se passer en cours ?
Qu'est-ce que je fais ? Qu'est-ce que vous faites ?

☞ différence entre *information* (ou même « *recevoir de l'information* ») et *connaissance*

Comment l'information devient connaissance ?

Les deux processus de l'apprentissage :

- l'instruction/l'enseignement (information)
- **l'appropriation** (transformation de l'information en connaissance)

Enseigner, c'est mettre un savoir à disposition. Apprendre, c'est se l'approprier.

☞ Si vous utilisez une IA, vous ne vous **appropriez** rien du tout : il ne reste finalement pas grand chose

Jean-Cédric Chappelier
& Jamila Sam

Bien apprendre à programmer / Rôle de l'IA

[Une partie du matériel des 4 slides suivants est empruntée à mes collègues J. Sam, C. Salzmann et S. Doeraene]

Pour apprendre, il faut **PRATIQUER par soi-même**
(y compris le debugging)

Se confronter aux difficultés, y réfléchir, reformuler, etc.
sont nécessaires pour (bien) apprendre

☞ Lire une solution (IA ou autre), sans avoir au préalable **FAIT par soi-même**, ne sert à rien

« *On a l'impression de devenir bête.*
On ne réfléchit plus, car on peut demander directement à une IA. »

Jean-Cédric Chappelier
& Jamila Sam

Utilisation de GenAI

Certes, les outils à base d'IA générative (ChatGPT, Github-Copilot, Claude, etc.) révolutionnent désormais l'approche au codage
et sont déjà considérés, à juste titre, comme des supports indispensables

Mais...

1. ils ne sont efficaces qu'**en complément d'un socle *minimal* acquis par la formation**

☞ un utilisateur qui ne saurait pas lire le code produit s'expose à intégrer des erreurs subtiles ou à construire des solutions qu'il ne pourra pas maintenir.

Jean-Cédric Chappelier
& Jamila Sam

Utilisation de GenAI et apprentissage

Certes, les outils à base d'IA générative (ChatGPT, Github-Copilot, Claude, etc.)
révolutionnent désormais l'approche au codage
et sont déjà considérés, à juste titre, comme des supports indispensables

Mais...
2. ils ne sont pas faits pour l'apprentissage !

(cf https://www.epfl.ch/education/teaching/index-html/ai-teaching/ai-in-student-learning/)

> Utiliser une calculatrice vous apprend-il les bases du calcul ?
>
> Pouvez-vous comprendre pourquoi une calcul (sur calculette) est faux sans connaître les bases de l'arithmétique ?

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

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

Vous n'avez pas encore acquis toutes les bonnes méthodes de travail.
Utiliser des outils genAI pour ***croire*** de gagner du temps, vous ne développez ni votre efficacité, ni vos méthodes d'apprentissage et d'analyse pertinentes.

☞ On veut former des ingénieur(e)s, pas juste des codeuses/codeurs.

Jean-Cédric Chappelier
& Jamila Sam

Pourquoi les IA disent des bêtises (surtout aux débutant(e)s)

Étudiant(e)s :

- question (« *prompt* ») mal posée :
  mauvais ou manque de contexte, mauvaise compréhention de l'erreur
- manque de recul critique pour évaluer la pertinence de la réponse
  ☞ gardez un **esprit critique**

GenAI :

- mélange de différents langages de programmation
  (en raison de fortes similitudes mais subtiles différences)
- ne compile pas et n'exécute pas le code, ne fait que le **prédire**
  (prédit la suite la plus probable suivant ses données d'entraînement)
- génération d'erreurs par reformulation (p.ex. troncature de parties de code)

Jean-Cédric Chappelier
& Jamila Sam

Conseils d'utilisation d'IA dans ce cours (1/2)

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.

Conseil : Utilisez les IA (Chatbots) pour chercher des réponses à vos questions **une fois que vous avez fait l'effort de chercher par vous-même**
mais en tout cas **pas** pour produire du code ni pour déboguer votre code

☞ https://www.epfl.ch/education/teaching/index-html/ai-teaching/ai-in-student-learning/

Conseils d'utilisation d'IA dans ce cours (2/2)

Concrètement :

✔ Acceptable :

- comprendre un message d'erreur obscur du compilateur,
  une fois que vous avez fait (plusieurs fois) ***l'effort*** **d'essayer de le comprendre par vous-même**
- proposer de nouveaux exercices sur un sujet donné

 Possible mais **dangereux** :

- recherche initiale d'informations sur un sujet inconnu

✘ Vous devriez vous **interdire** à vous-même :

- produire du code ou une solution d'exercice
- que ce soit votre premier réflexe lorsque vous êtes face à un obstacle

Jean-Cédric Chappelier
& Jamila Sam

Le langage C++

Le langage C++ est un langage orienté-objet compilé fortement typé.
Schématiquement :

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 ;
- 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.

Jean-Cédric Chappelier
& Jamila Sam

Variables et types

À 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).

Jean-Cédric Chappelier
& Jamila Sam

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 `\'` (donc `'\''`)
  le caractère `\` se représente par `\\` (donc `'\\'`)
- 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.

Jean-Cédric Chappelier
& Jamila Sam

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**

Jean-Cédric Chappelier
& Jamila Sam

« Etude de cas » (une question en fait)

Qu'affiche le code suivant :

```
double x(3 / 4);
cout << x << endl;

double a(3);
double b(4);
double y(a / b);
cout << y << endl;
```

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)
```


Jean-Cédric Chappelier
& Jamila Sam

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);
```
(mais `constexpr` serait encore mieux)

```
double const interet(3.0*taux+1.5);
```
(ici `constexpr` est impossible (sauf si `taux` est lui-même `constexpr`))

```
double const g(9.81);
```
(ici aussi `constexpr` serait encore mieux)

Jean-Cédric Chappelier
& Jamila Sam

Expressions constantes

Depuis 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 (c.-à-d. que 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`.

Jean-Cédric Chappelier
& Jamila Sam

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;
```

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