Magistère de Physique Fondamentale d'Orsay L3 2024-2025

Informatique → C

Bartjan van Tent, bartjan.van-tent@ijclab.in2p3.fr

Site du cours :
`https://hebergement.universite-paris-saclay.fr/mpo-informatique/`

Plan

1. Introduction & motivation, bases Linux & Emacs
2. Premier programme en C
3. Variables et opérateurs
4. Tests et boucles
5. Entrées, sorties, fichiers, graphiques
6. Fonctions
7. Pointeurs
8. Tableaux dynamiques
9. Générateur de nombres aléatoires
10. Résolution d'équations différentielles
11. Compléments de cours : structures, création de bibliothèques

1. Motivation

Pourquoi un cours de programmation pour les physiciens ?

Parce que dans beaucoup de domaines de physique la programmation est devenue indispensable, par exemple pour :

- la résolution numérique des équations différentielles,
- l'analyse des données d'une expérience,
- la production de simulations pour faire des tests.

Pourquoi le **langage C** ?

Parce que c'est un langage puissant, rapide et beaucoup utilisé :

- Rapide car **langage compilé** (= compilé entièrement en une seule fois avant l'exécution) et non pas interprété (= compilé ligne par ligne pendant l'exécution, comme le Python par exemple).
  Compilation = traduction d'un programme écrit dans un langage de programmation (compréhensible par les humains) en langage machine (exécutable par l'ordinateur).
- Beaucoup utilisé, non seulement dans la recherche scientifique, mais aussi dans d'autres domaines. Par exemple, le système d'exploitation Windows et le compilateur du langage Python ont été programmés en C.
- Puissant car on peut faire tout et contrôler tout.

Toutefois, certaines choses ne sont pas faciles à faire en C.

- L'extension **C++** (enseigné en M1) ajoute des méthodes au C pour simplifier certains aspects ou programmer d'une autre façon.
- Si la rapidité et le contrôle de tout ne sont pas de soucis, le **Python** (option au 2nd semestre) est une alternative intéressante et plus facile.

On peut aussi combiner Python et C pour avoir le meilleur des deux mondes !

Ce cours est un cours du **C strict**. Cependant, pour vous simplifier la vie, on va utiliser les méthodes du C++ pour l'interaction avec le clavier, l'écran et les fichiers, et le Python (matplotlib) pour faire des graphiques.

Histoire du C

- Développé en 1972 par Dennis Ritchie (Bell Labs) avec comme but principal le développement du système d'exploitation Unix.
- Devient populaire en 1978 quand le livre *"The C Programming Language"* de Kernighan et Ritchie est publié (et est resté populaire depuis).
- Origine du nom : le 'C' n'est pas une abbréviation, il s'appelle comme ça parce qu'il est basé sur un langage B (développé par Ken Thompson, Bell Labs, en 1969), qui lui était une version simplifiée du BCPL (*Basic Combined Programming Language*, developpé par Martin Richards, Université de Cambridge, en 1966).

(C++ : Bjarne Stroustrup, Bell Labs, 1983 ; Python : Guido van Rossum, Centre de Mathématiques et d'Informatique Néerlandais, 1991.)

Organisation du cours

Ce cours consiste en trois parties :

- Le cours en amphi (1er & 2nd semestre, périodes 2 & 3)
- Les TD (1er & 2nd semestre, périodes 2 & 3)
- Le projet informatique (2nd semestre, périodes 3 & 4)

Modalités de contrôle des connaissances :

- Examen écrit concernant le cours et les TD en mai
- Examen oral individuel concernant le projet en avril

Supports de cours, tous disponibles sur le site du cours :

- Les transparents du cours
- Le poly de cours (écrit par François Naulin, mon prédecesseur)
- Le poly avec les exercices de TD et une liste de commandes Linux et Emacs
- Notice sur l'utilisation de Python/matplotlib pour faire des graphiques

Niveau du cours : accessible pour tout le monde, aucune expérience de programmation réquise. Néanmoins, à la fin du cours vous saurez programmer des calculs intéressants.

Linux

Le **système d'exploitation** Linux est une alternative à Windows, généralement plus stable et plus sûr que ce dernier. En plus, il est gratuit et beaucoup de logiciels sont gratuits et faciles à installer. Origine du nom : version de Unix pour PC, développé initialement par Linus Torvalds en 1991.
Différence en pratique : même s'il existe la possibilité de faire beaucoup en cliquant avec la souris, on travaille généralement plus avec une fenêtre dite '**terminal**' où l'on tape ses commandes en utilisant le clavier.

Liste de quelques commandes :

ls (*list*) : afficher le contenu d'un répertoire. Ex. : `ls` — affiche tout le contenu (les fichiers) du répertoire courant.

cp *source destination* (*copy*) : faire une copie d'un fichier.
Ex. : `cp prog1.cpp prog2.cpp` — fait une copie, appelée 'prog2.cpp', du fichier 'prog1.cpp'.

mv *source destination* (*move*) : déplacer ou renommer un fichier.
Ex. : `mv prog1.cpp prog2.cpp` — le fichier 'prog1.cpp' est renommé 'prog2.cpp'.

rm *nom* (*remove*) : supprimer un fichier. Ex. : `rm prog1.cpp` — supprime le fichier 'prog1.cpp'.

ccc *nom* (commande spécifique aux ordinateurs du magistère) : compiler et exécuter un programme C. Ex. : `ccc prog1.cpp` — Compile et ensuite exécute le programme 'prog1.cpp'.

Vous avez tous un **répertoire** initial personnel. C'est le répertoire courant (= le répertoire où vous vous trouvez ; répertoire = dossier) quand vous ouvrez une fenêtre terminal. Pour bien organiser votre travail, il est conseillé de créer des sous-répertoires dans votre répertoire initial, p.ex. 'TD1', 'TD2', 'projet', etc. et de créer vos fichiers dans ces sous-répertoires.

Symboles spéciaux : ~ (tilde) — répertoire initial, . (point) — répertoire courant, .. (deux points) — répertoire père du répertoire courant.
Ex. : `mv ../prog1.cpp .` — déplace le fichier 'prog1.cpp', qui se trouve dans le répertoire père du répertoire courant, vers le répertoire courant (remarque bien le point comme destination, ainsi que l'utilisation du slash /).

mkdir *nom* (*make directory*) : créer un répertoire. Ex. : `mkdir TD1` — crée le sous-répertoire 'TD1' dans le répertoire courant.

rmdir *nom* (*remove directory*) : supprimer un répertoire. Ex. : `rmdir TD1` — supprime 'TD1' (qui doit être vide) dans le rép. courant.

cd *nom* (*change directory*) : se déplacer dans l'arborescence des répertoires. Ex. : `cd TD1` — on se déplace vers le sous-répertoire 'TD1' du répertoire courant, qui devient le nouveau répertoire courant ; `cd ..` — on revient vers le répertoire père, qui redevient le répertoire courant.

Pour vous faciliter la vie, utiliser les **touches flèches verticales** ↑, ↓ (historique des commandes tapées précédemment) et la **touche de tabulation** ↹ (complétion de noms de fichiers à partir des premiers caractères tapés).

Emacs

L'**éditeur de texte** Emacs (Stallman & Steele, 1976 ; origine du nom : *editor macros*) est un éditeur très puissant et sera utilisé en TD pour écrire les programmes. On le lance avec la commande `emacs prog1.cpp &`.

- Cela ouvre le fichier 'prog1.cpp' s'il existe, ou crée un nouveau fichier appelé 'prog1.cpp' sinon. Le fichier s'ouvre dans une nouvelle fenêtre.
- L'esperluette & à la fin de la commande fait tourner Emacs en arrière-plan, ce qui libère la fenêtre terminal pour d'autres commandes. Sans esperluette la fenêtre terminal resterait bloquée jusqu'à la fermeture d'Emacs.
- La première partie du nom du fichier est votre choix, mais les programmes C doivent obligatoirement avoir une extension '.cpp' à la fin du nom.

Liste de quelques raccourcis clavier utiles dans Emacs :

Ctrl-x Ctrl-s	sauvegarder
Ctrl-x Ctrl-c	quitter
Ctrl-espace	marque la position actuelle du curseur comme le début d'une sélection
Alt-w	copier la sélection (entre le début marqué par Ctrl-espace et la position actuelle du curseur)
Ctrl-w	couper la sélection
Ctrl-y	coller la sélection précédemment copiée ou coupée

2. Premier programme en C

```
#include<iostream>   /* Message pour le pre-compilateur (#) :
                        inclure bibliotheque pour cout (C++) */
using namespace std; /* Pour utiliser les fonctions C++ plus
                        facilement (cout au lieu de std::cout) */
int main() {  /* Fonction main,
                 dont les accolades delimitent le contenu */
  int i;      // Declaration de variable
  i = 5;      // Affectation d'une valeur a la variable
  i = i + 2;  // Affectation d'une nouvelle valeur a la variable
  cout << "La valeur de i est : " << i << endl; // Affichage
  return 0;   // Pour indiquer que le programme s'est bien deroule
}
```

Résultat après compilation et exécution avec la commande Linux *ccc* : le message "La valeur de i est : 7" est affiché à l'écran.

- On peut rajouter du **commentaire** à un programme de deux façons : // et tout ce qui suit sur une ligne est ignoré par le (pré-)compilateur, ainsi que tout ce qui se trouve entre /* et */
- Les lignes commençant par # sont des instructions pour le pré-compilateur. `#include<iostream>` dit au pré-compilateur de mettre les définitions de la **bibliothèque** `iostream` à cet endroit, afin que le compilateur qui vient après connaisse la commande `cout`.

- Tout programme en C doit avoir une fonction du type `int` qui s'appelle '**main**' et n'a pas d'arguments (parenthèses vides), qui contient le programme principal.
- Les **accolades** délimitent un bloc de commandes qui appartiennent ensemble à ce qui précède (ici la fonction main). Le contenu d'une fonction se trouve obligatoirement entre accolades.
- Toutes les commandes se terminent obligatoirement par un **point-virgule**. On peut étaler une commande sur plusieurs lignes ou mettre plusieurs commandes sur une ligne.
- En C il faut obligatoirement **déclarer** explicitement toutes les **variables** qu'on utilise. Ici la seule variable du programme s'appelle '`i`' et est du type entier (*integer*).
- La déclaration d'une variable réserve une petite boîte dans la mémoire de la bonne taille pour mettre une valeur de ce type, mais n'y met **aucune valeur d'initialisation** ! Après `int i;` la valeur de `i` est aléatoire.
- Une commande d'**affectation**, indiqué par l'opérateur d'affectation =, calcule d'abord le résultat de l'expression à droite, puis affecte cette valeur à la variable à gauche, écrasant la valeur précédente de la variable.
- La commande C++ '**cout**' (*character out*) affiche ce qui suit à l'écran, soit des chaînes de caractères (entre guillemets doubles à l'anglaise) de façon littérale, soit des valeurs de variables. `endl` (*end line*) fait aller à la ligne suivante. Tous les éléments sont séparés par des <<.

Tout dans l'ordre

L'exécution d'un programme commence au début de la fonction `main` et les commandes sont exécutées strictement dans l'ordre. En particulier, n'utilise jamais des variables sans les avoir initialisées !

Par exemple, le bout de programme `int i,n; i=n; n=4;` ne marche pas correctement. D'abord `i` et `n` sont déclarés, mais ont encore une valeur quelconque. Puis la valeur de `i` devient celle de `n`, donc c'est toujours une valeur quelconque. Enfin `n` devient égal à 4, mais la valeur de `i` ne change pas ! Il fallait plutôt écrire `int i,n; n=4; i=n;`, maintenant `i` et `n` sont tous les deux égaux à 4.

Indentation

Tout comme le commentaire, l'indentation des lignes est une façon de rendre un programme plus lisible et plus compréhensible pour les humains, sans changer quelque chose pour le compilateur. C'est fortement conseillé !

Chaque bloc de commandes (entre accolades) est indenté de deux espaces. Emacs connait ces règles d'indentation (quand le fichier a l'extension .cpp) et les appliquera automatiquement pour une nouvelle ligne. On peut aussi utiliser la touche de tabulation (n'importe où dans la ligne) pour indenter cette ligne de la façon correcte selon Emacs. Cela permet aussi de trouver des erreurs : si l'indentation d'Emacs n'est pas correcte, vous avez probablement oublié un point-virgule ou une parenthèse sur la ligne précédente.