Programmation et algorithmique

IN101

ENSTA Paris - TC 1ère année

François Pessaux

U2IS

2022-2023

prenom.nom@ensta-paristech.fr

Préambule

Avant de commencer. . .

Moi == François Pessaux
U2IS, bureau R223

- Ne pas hésiter à poser des questions, même en amphi.
- Posez votre question que vous jugez « bête » : d'autres ici ont la même.
- Ne pas hésiter à dire *« M'sieur, vous vous z'êtes pas trompé sur xyz ? »*
- Mon bureau et mon mail sont facilement accessibles.
- Terminez et conservez vos exercices, vos programmes.
- Utilisez `git` pour conserver les modifications.
  - GitLab asso DaTA : `https://gitlab.data-ensta.fr/`

Pourquoi dois-je subir des cours d'informatique ?

- Numérique [1] présent dans tous les domaines, tous les métiers.
- Ingénieurs de formation généraliste.
- Pour vos cours futurs (vue pragmatique à court terme).
- Pour votre future carrière :
  - Simulation numérique (mécanique, finance, etc.) ? ⇒ info + ...
  - Systèmes embarqués ? ⇒ info + ...
  - Data-science ? ⇒ info + ...
  - Cyber-sécurité ? ⇒ info + ...
  - Chef de projet ? ⇒ info + ...
  - Etc.

1. et pas « digital », qui est relatif au doigt !

Vous avez principalement déjà appris...

- En classes préparatoires :
  - Un langage de programmation : `Python`.
  - Utiliser la bibliothèque mathématique de `Python` (`NumPy`).
  - Comment programmer la résolution de problèmes mathématiques.
  - Écrire des programmes pour des problèmes relativement modestes.
- En IN102 :
  - Un langage de programmation : `C`.
  - Écrire des programmes pour des problèmes relativement modestes.

Buts de ce cours

- Apprendre à concevoir des algorithmes.
- Apprendre à se poser des questions sur la spécification d'un programme (ce qui est demandé ou ce que l'on veut faire).
- Acquérir une démarche systématique et scientifique :
  1. d'analyse,
  2. de formalisation,
  3. de conception,
  4. d'implantation.
- Implanter dans un langage.

Pourquoi ces buts ?

- Être prêt pour des problèmes plus complexes.
- Être capable de concevoir des algorithmes maintenables.
- Être capable de concevoir des algorithmes passant à l'échelle.
- Démarche applicable dans de nombreux domaines.

Attention !

Pratiquez !

Juste réviser 3 jours à l'avance ne suffira pas si vous ne vous êtes pas exercés (idem pour IN103, IN104) !

Prenez le temps de réfléchir avant de coder (idem pour IN103, IN104) !

Même (surtout ?) en examen, ce n'est pas du temps perdu !

Architecture

« Vous avez dit ordinateur ? »

- *Mais pour quoi faire ?* Pour traiter de l'information.
- « *Information ?* »
  - C'est un message.
  - Écriture avec des symboles : code (injection $S_1 \to S_2^+$).
  - Monde numérique : code binaire.
- Architecture d'un ordinateur :
  - Mémoire(s) : pour stocker de l'information.
  - Microprocesseur (CPU) : pour travailler l'information.
  - Périphériques : pour échanger avec « *l'extérieur* ».
- « *Traitement* » $\Rightarrow$ description de « *comment* » : programme.
- Architecture de Von Neumann :
  - Le programme est stocké dans la mémoire...
  - ... dans la même mémoire que les données (informations).

La mémoire

- Pourquoi en avoir besoin ?
  - Certains traitements peuvent se faire directement : $x \mapsto x+1$.
  - Intervention de la notion de « *temps* » : exemple, un compteur.
  - $\Rightarrow$ Besoin de se rappeler de (d'un) « *l'état* » précédent.
- Différents fonctionnements de mémoire :
  - RAM (Random Access Memory) : lecture / écriture – Volatile.
  - ROM (Read Only Memory) : lecture – Persistante.
  - EPROM (Erasable Programmable Read Only Memory) : lecture (/ écriture mais en mode de fonctionnement particulier) – Persistante.
- Différents types de mémoire :
  - Mémoire de masse (disques mécaniques) : ~ 10 ms
  - Mémoire de masse (disques SSD) : $\sim 0.1 - 0.3$ ms
  - Mémoire vive (RAM) externe au CPU : $\sim 40 - 50$ ns
  - Mémoire cache $\pm$ interne au CPU : $\sim 5 - 10$ ns
  - Registres : ~ 1 ns

Le microprocesseur

- C'est l'unité qui *« travaille »*.
- Idéalement segmentée en 2 parties :
  - L'unité arithmétique et logique (UAL / ALU) : effectue les calculs.
  - L'unité de contrôle : décodage et séquencement des instructions.
- Contient de la mémoire très rapide (~ 1 ns) : les registres.
- *« Instruction »* : opération élémentaire effectuée par le CPU (addition de 2 registres, copie registre $\leftrightarrow$ mémoire, registre $\leftarrow$ registre, décalages, sauts ...
- Chaque CPU (x86, ARM, amd64 ...) est différent, a son propre jeu d'instructions.

Le langage

Quel langage ?

C

- Car :
  - Déjà vu par vous.
  - Permet une pratique prolongée du langage.
  - Permet l'implantation dans n'importe quel autre langage.
- Constructions nécessaires et suffisantes pour ce cours :
  - Expressions de base (arithmétiques, conditionnelles, variables).
  - Boucles (`for`, `while`).
  - Fonctions.
  - Tableaux et allocation dynamique.
  - Lecture/écriture de fichiers.

Quelques rappels – compiler/exécuter

- Compilation : `gcc -Wall -Werror -Wfatal-errors hello.c`
- Exécution : `./a.out`
- Compilation :
  `gcc -Wall -Werror -Wfatal-errors hello.c -o hello.x`
- Exécution : `./hello.x`
- Surtout pas : ~~`gcc hello.c -o hello.c`~~
  - ⇒ détruirait votre fichier source.
- `-Wall` indispensable pour vérifier l'absence de *warnings*.
- `-Werror` indispensable pour considérer les *warnings* comme erreurs.
- `-Wfatal-errors` indispensable pour arrêter la compilation à la première erreur.

⚠ *Warnings* (sans `-Werror`) en examen ⇒ pénalité sur l'exercice.

⚠ Ne compile pas en examen ⇒ pénalité sur l'exercice.

Quelques rappels – syntaxe

- Variables déclarées avec leur type.
  - **unsigned int** age = 18 ;
- Fonctions : type-retour nom-fonction (type-argument, ...) { corps }
  - **char*** make_string (**unsigned int** len, **char** init) { ... }
- Toujours 1 et 1 seule fonction `main`
  - **int** main (**int** argc, **char** *argv []) { ... }
- Types de base : **char**, **int**, **float**, **double**.
  - Modificateurs : **signed**, **unsigned**, **short**, **long**.
- Opérateurs :
  - Arithmétiques : +, −, /, *, %
  - Comparaisons : ==, !=, <, >, <=, >=
  - Logiques : &&, ||, !
  - Affectation et in/décrémentation : =, +=, −=, ..., ++, −−
  - Pointeurs : &, *

Quelques rappels – structures de contrôle

- Boucles :
  - **while** (...) { ... }
  - **do** { ... } **while** (...) ;
  - **for** (...; ...; ...) { ... }
- Conditionnelles :
  - **if** (...) { ... }
  - **if** (...) { ... } **else** { ... }
  - **switch** (...) { **case** ... : ... **break** ; **case** ... : ... **break** ; }

Quelques rappels – sortie à l'écran : `printf`

- Nécessite : **#include** <stdio.h>
- printf ("Eat at Joe's\n") ;
- printf ("%d = %d + %d\n", x + y, x, y);
- Format = chaîne de caractères contenant (ou pas) des séquences %♪.

%d	un **int**
%ld	un **long int** en décimal
%u	un **unsigned int** en décimal
%x	un **int** en hexadécimal
%f	un **float**
%lf	un **double**
%e	un **double** en notation scientifique
%.7lf	un **double** avec 7 chiffres après la virgule
%07d	un **int** en décimal sur 7 digits (remplissement frontal avec des 0)
% 7d	un **int** en décimal sur 7 digits (remplissement frontal avec des ' ')
%c	un **char** (comme caractère ASCII, pas comme entier)

Quelques rappels – entrée au clavier : scanf

- Nécessite : **#include** <stdio.h>
- Lecture au clavier selon le format spécifié (séquences %♪).
- Si saisie pas en accord avec ce qu'attend le format : imprévisible.
- Format avec uniquement des % (pas de « *message* »).
- Passage par adresse des variables réceptrices.

input.c

```c
#include <stdio.h>

int main ()
{
  int i, j ;
  scanf ("%d %d", &i, &j) ;
  printf ("i: %d, j: %d\n", i, j);
  return (0) ;
}
```

```
$ gcc -Wall input.c -o input
$ ./input
45 67
i: 45, j: 67
$ ./input
5
FHG
i: 5, j: 0
$ ./input
DFG 6
i: 0, j: 0
```

Quelques rappels – arguments de ligne de commande

- **int** main (**int** argc, **char** *[] argv) :
  - `argc` : nombre d'arguments + 1 :
  - `argc` : tableau des chaînes passées en arguments.
  - `argc[0]` : nom de l'exécutable courant.
    - « Vrais » arguments commencent après.
- Besoin de transformer chaîne → `int`, `float`, ...
  - Nécessite **#include** <stdlib.h>
  - `atoi` string → int
  - `atol` : string → long int
  - `atoll` : string → long long int
  - `atof` : string → float

Quelques rappels – tableaux et allocation dynamique

- Allocation statique
  - Si taille connue à la compilation et relativement modeste.
  - Si tableau non retourné par la fonction.

```
#define SIZE (20)
char name[SIZE] ;

void f () {
  int t[5] ;
  ...
}
```

- Allocation dynamique sinon.
  - Allocation : malloc.
  - Libération : free.

```
double* make_array (int size) {
  double *t ;
  t = malloc (size * sizeof (double)) ;
  if (t == NULL) { ... /* Gérer l'erreur. */ }
  ...
  return (t) ;  /* Libérer avec free ultérieurement. */
}
```

Algorithme

Ce que n'est pas un algo . . .pour vous

- Pas une recette de cuisine, dans tous les sens du terme :
  - 1er degré : les instructions sont à suivre par la machine
    - à vous de trouver quelles instructions doivent être faites.
  - 2nd degré : pas un tas de calculs infâmes en lequel on espère plutôt que l'on ne croit.
- Pas un langage de programmation
  - Langage = juste des mots pour décrire un algorithme.

Ce qu'est un algorithme

- Méthode de passage d'un ensemble d'entrées E à un ensemble de résultats R.
- Cette méthode s'appuie sur du calcul.
- ⇒ Fonction (qui peut être partielle).

LA première question...

« *Qu'est-ce que j'ai, qu'est-ce que je veux ?* »

1. Les données à traiter : « entrées », hypothèses du problème.
2. Les résultats à obtenir : « sorties ».

Exemple : « *Je veux calculer une racine carrée.* »

- En entrée j'ai un « nombre » x.
- En sortie j'ai un « nombre » y tel que $y^2 = x$ ... ou pas.

C'est après que l'on se demande comment le faire.

Liens entre entrées et sorties

Liens entrées → sorties :

1. Celui exprimé par le futur utilisateur : spécification.
2. Celui décrit par le calcul : algorithme puis programme.

Un algorithme (puis un programme) « fonctionne bien » ssi 2 satisfait 1.

« Je veux additionner 2 entiers. » (1)

foo.c

```c
#include <stdio.h>

int addition (int x, int y) {
  return (x + y) ;
}

int main () {
  printf ("%d\n%d\n", addition (0, 0), addition (4, 6)) ;
  return 0 ;
}
```

```
$ gcc -Wall foo.c
$ ./a.out
0
10
```

Ça marche ☺.

« Je veux additionner 2 entiers. » (2)

foo.c

```c
#include <stdio.h>

int addition (int x, int y) {
  int res = x ;
  for (int i = 0; i < y; i++)
    res = res + 1 ;
  return res ;
}

int main () {
  printf ("%d\n%d\n", addition (0, 0), addition (4, 6)) ;
  return 0 ;
}
```

```
$ gcc -Wall foo.c
$ ./a.out
0
10
```

Ça marche ☺. C'est plus « compliqué », mais ça marche.

« Je veux additionner 2 entiers. » (3)

foo.c

```c
#include <stdio.h>

int addition (int x, int y) {
  return (x * y) ;
}

int main () {
  printf ("%d\n%d\n", addition (0, 0), addition (4, 6)) ;
  return 0 ;
}
```

```
$ gcc -Wall foo.c
$ ./a.out
0
24
```

- Optimiste : « *Ça marche presque : pour addition (0, 0) c'est bon.* »
- *Pessimiste :* « Ça marche presque pas, j'ai seulement 1 cas correct. »
- Réaliste : « *Ça ne marche pas.* »

Tester : ⇒ plusieurs cas.