IN103
Résolution de problèmes algorithmiques
une approche fondée sur les structures de données

Alexandre Chapoutot

Année académique 2023-2024

`https://perso.ensta-paris.fr/~chapoutot/teaching/in103/`

Avant de commencer

Moi == (encore) Alexandre Chapoutot
U2IS, bureau R217

- 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.

Remise en contexte

Pour le moment vous avez étudié, par ordre chronologique

- IN102 : la programmation en langage C (maitriser la syntaxe de C)
- IN101 : les bases de l'algorithmique (étudier et mettre en oeuvre une méthodologie de conception des algorithmes)

En IN103, nous poursuivons l'apprentissage de la pensée algorithmique :

- en étudiant des algorithmes connus ;
- en se fondant sur une organisation des données adaptées => **structures de données**.

Pour cela nous utiliserons :

- les traits caractéristiques du langage C, notamment les pointeurs ;
- une bibliothèques de structures de données conçue spécialement, nommée **libin103**.

Pourquoi une bibliothèque spécifique au cours ?

- Par ce que le langage C n'est pas doté d'une bibliothèque de structure de données comme d'autres langages (p. ex., C++, OCaml, Java, ...)
- Il existe des bibliothèques de structures de données en C
  - GLib (associée à GNOME)
  - Gnulib
  - SGLIB
  - liblfds
  - et plein de trucs sur GitHub

  Mais elles ne sont pas aisées à prendre en main ou incomplètes.
- La bibliothèque **libin103** a été concue pour simplifier l'utilisation et illustrer tous les concepts du cours, pas pour les performances

Remarque : L'utilisation d'une bibliothèque en IN103 vous facilitera la prise en main de bibliothèques dans d'autres langages, en particulier, le langage C++ étudié en 2A.

Déroulement du cours et évaluation

Sauf exception, les séances sont composées

- 1h de cours magistral (avec beaucoup d'informations)
- 2h de TD/TP (avec encore beaucoup d'informations)

et $x \gg 0$ secondes de travail à la maison

La note sera composée

- Notes de QCM hebodmadaires ;
- Note de l'examen final

en suivant la règle pour les personnes sans aménagement :

$$\text{note} = 0.2 \times \text{QCM} + 0.8 \times \text{Examen}$$

ou en suivant la règle pour les personnes ayant un aménagement :

$$\text{note} = 1.25 \times (0.2 \times \text{QCM} + 0.8 \times \text{Examen})$$

Quelques conseils pour réussir

Mantra de l'apprentissage en IN103 (mais pas que)

- Lire n'est pas comprendre
- Copier-coller n'est pas comprendre
- Écrire est comprendre

Méthode de travail (à appliquer régulièrement)

- **Réécrivez**/Compilez/Exécutez les corrections des exercices
- **Questionnez vous** sur les lignes qui sont utlisées pour réaliser le programme (pourquoi comme ça ?, est-ce qu'il y a une autre façon de faire ?, ...)
- **Essayez d'expliquer à voix haute** les programmes que vous (ré)écrivez au canard en plastique, cf MO101 ;-) ou à un ou une camarade

La clef de la réussite : s'entraîner, s'entraîner, s'entraîner, ...

Agenda

- 23 janvier 2024 : **Quelques rappels sur le langage C et un peu plus**
  - Compilation séparée, make, Pointeurs
- 30 janvier 2024 : **Structures de données linéaires**
  - Listes chaînées, Piles, Files, Ensembles
- 6 février 2024 (3H TD/TP) : **Structures arborescentes – partie 1**
  - Arbres binaires, Arbres binaires de recherche
- 13 février 2024 : **Structures arborescentes – partie 2**
  - Tas, Ensembles disjoints (union-find)
- 27 février 2024 : **Graphes – partie 1**
  - Représentations, Parcours de graphes (largeur/profondeur, tri topologique)
- 5 mars 2024 : **Graphes – partie 2**
  - Composantes fortement connexes, Plus court chemin, Arbre couvrant de poids minimum
- 19 mars 2024 : **Examen** (3h)

Remarque : Le contenu des séances changera peut-être en fonction des envies de l'enseignant

Plan du cours

Compilation séparée

Rappels sur le langage C – 1

Principales constructions du langage :

- `<instruction>;`
  Exemples : affectation `x = y + 3;` ou appel de fonction `f(x);`
- `{ <instructions> }` – bloc d'instructions
  Exemple : `{ int x = 2; int y = x + 3; }`
- `if (<condition>) <instruction(s)> else <instruction(s)>`
  Exemple :
  ```
  if (x % 2 == 0) { printf ("pair\n"); } else { printf ("impair\n"); }
  ```
- `while (<condition>) <instruction(s)>`
  Exemple : `int x = 10; while (x > 0) { printf ("%d\n", x); x--;}`
- `for (<init>; <condition>; <step>) <instruction(s)>`
  Exemple : `for (int x = 0; x<10; x++) { printf ("%d\n", x); }`
- `switch (<exp>) { case: <instruction(s)>; ... defaut: <instruction(s)>; }`
  Exemple :
  ```
  switch (c) { case 'q': return 0; default: printf ("joue␣encore\n"); }
  ```

Rappels sur le langage C – 2

- **Types de données de base**

Nom	Taille	Description
`char`	1 byte	caractère dans l'encodage ASCII, `'a'`, `'\n'`
`int`	4 bytes	valeur entière signée, `97`, `0xFF`, `0777`
`long long int`	8 bytes	valeur entière signée de grande taille
`float`	4 bytes	valeur flottante simple précision
`double`	8 bytes	valeur flottante double précision

- **Les principales opérations**

Symboles	Description
`+, -, *, /, %`	Opérations arithmétiques
`==, <=, >=, !=`	Opérations de comparaison
`&&, ||, !`	Opérations booléennes
`?:`	Expression conditionnelle (`x >= 0 ? "POS": "NEG"`)
`i++, i--`	Post incrémentation/décrémentation

Les types

Information essentielle pour une variable : donne l'espace mémoire nécessaire pour la stocker (`sizeof`) et les opérations admissibles pour la manipuler.

Rappels sur le langage C – 3

- **Types de données composites**

Nom	Description
`struct`	Structure regroupant plusieurs variables de types potentiellement différents dans un même type. (`struct point {double x; double y};`) et accès avec la notation pointée (`struct point p; p.x = 2.0;`)
`enum`	Structure regroupant plusieurs constantes entières dans un même type. (`enum color {BLACK, WHITE};`) et (`enum color col = WHITE;`)
`type a[SIZE]`	Tableau statique de SIZE éléments du même type `type`. (`int a[3] = {1, 2, 3};`) et (`a[0]`) est le premier élément et (`a[2]`) est le dernier élément.

- **Les chaînes de caractères**
  Ce sont des tableaux de caractères terminés par le caractère `'\0'`.

Rappels sur le langage C – 4

Les pointeurs sont déclarés grâce à la présence du caractère `*` à droite du type lors de la déclaration de variable

- `int *a;` *a* est un pointeur sur un entier ;
- `char *b;` *b* est un pointeur sur un caractère ;
- `int *c[2];` *c* est un tableau de pointeurs sur des entiers ;
- `struct point *p;` *p* est un pointeur sur une structure de type `point`.

Remarque : Pointeurs = Adresses mémoires

Manipulation des pointeurs :

- `a` représente l'adresse de l'entier pointé ;
- `*a` représente la valeur stockée à l'adresse (**déréferencement**) ;
- `int x; a = &x;` & donne l'adresse d'une variable ;
- `p->x = 2.0;` identique à `(*p).x`, raccourci pour déréfencer un pointeur sur une structure.

Rappels sur le langage C – 5

Tableaux

- **Allocation statique**
  - si la taille est connue à la compilation et de taille modeste ;
  - si le tableau n'est pas valeur de retour d'une fonction.

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

- **Allocation dynamique** dans les autres situations
  - Allocation mémoire : `malloc`
  - Libération mémoire : `free`

```
double *array = malloc (10 * sizeof(double));
if (array == NULL) { /* Erreur \‘a traiter */ }
```

Rappels sur le langage C – 6

Une fonction est décrite par

- un **prototype** (ou signature) donnant le type de la valeur retournée, son nom et le type de ses paramètres ;
- un **bloc d'instructions**.

```c
double sqr (double x) {
  return x * x;
}
```

- **Prototype :**
  `double sqr (double)`
- **Bloc d'instructions :**
  `{ return x * x; }`

Fonction principale

Un programme écrit en langage C a toujours une unique fonction principale nommée `main` dont le prototype complet est `int main (int, char**)`.

Bibliothèque standard de C

Ensemble de fonctions organisées par catégories dont

- Gestion de la mémoire et utilitaires (`#include <stdlib.h>`)
  Fonctions : `malloc`, `free`, `atoi`, `rand`, ...
- Entrées/Sorties (`#include <stdio.h>`)
  Fonctions : `printf`, `scanf`, `fopen`, `fclose`, `fgets`, `fputs`, `perror`, ...
- Chaînes de caractères (`#include <string.h>`)
  Fonctions : `strlen`, `strncpy`, `strncmp`, ...
- Mathématiques (`#include <math.h>`)
  Fonctions : `sin`, `cos`, `pow`, `ceil`, `floor`, ...
- Classification des caractères (`#include <ctype.h>`)
  Fonctions : `isalpha`, `isupper`, `isspace`, `isdigit`, ...
- Valeurs booléennes (`#include <stdbool.h>`)
  Constantes : `true`, `false`

Anatomie d'un programme C – 1

```c
#include <stdio.h>

/* Defintions des fonctions auxiliaires */
double sqr (double x) {
  return x * x;
}

int main () {
  double x = 2.0;
  printf ("Sqr(%f) = %f\n", x, sqr(x));
  return 0;
}
```

- Une fonction doit être connue avant d'être utilisée dans la fonction principale.

Compilation / Exécution

- Compilation : `gcc -Wall -Werror sqr.c -o sqr.x`
- Exécution : `./sqr.x`

Anatomie d'un programme C – 2

```c
#include <stdio.h>

/* Declaration des fonctions auxiliaires */
double sqr (double x);  /* Notez ; terminal et l'absence de corps */

int main () {
  double x = 2.0;
  printf ("Sqr(%f) = %f\n", x, sqr(x));
  return 0;
}

/* Defintions des fonctions auxiliaires */
double sqr (double x) {
  return x * x;
}
```

- Une autre façon de faire est de déclarer les fonctions, c'est-à-dire, donner leurs prototypes ;
- Puis définir les fonctions, c'est-à-dire donner leur code.

Compilation séparée

Dans le cas de projets de programmation avec de nombreuses fonctions,

- il est commode de ne pas définir toutes les fonctions dans le fichier contenant la fonction principale.
  Conséquences : travail collaboratif et maintenance du code simplifiés
- il est également plus facile de partager du code entre différents projets si les fonctions ne sont pas toutes dans le même fichier.
  Conséquences : facilite la réutilisation

Exemple

```c
/* Fichier sqr.h */
double sqr (double x);

/* Fichier sqr.c */
double sqr (double x) {
  return x * x;
}
```

```c
/* Fichier main_sqr.c */
#include <stdio.h>
#include "sqr.h"

int main () {
  double x = 2.0;
  printf ("Sqr(%f) = %f\n", x, sqr(x));
  return 0;
}
```

Compilation séparée

Plusieurs fichiers :

- Les fichiers .c contenant les définitions des fonctions ;
- Les fichiers .h contenant les déclarations des fonctions qui sont exportées (mise à disposition) par le fichier .c ;

Convention: Sauf pour le fichier contenant la fonction principale, nous aurons des couples de fichiers .h/.c (**Ex.**, sqr.h / sqr.c)

Compilation

Pour l'exemple précédent, on peut compiler les fichiers un à un

- `gcc -Wall -Werror -c sqr.c`
- `gcc -Wall -Werror -c main-sqr.c -o main-sqr.o`
- `gcc -Wall -Werror main-sqr.o sqr.o -o main-sqr.x`

ou en forme condensée

- `gcc -Wall -Werror main-sqr.c sqr.c -o main-sqr.x`

Compilation séparée

Rappel : `#include` remplace la ligne par le contenu du fichier donné en argument.

- Si `#include "fichier.h"` cherche le fichier dans le répertoire courant puis dans les répertoires du système (**Ex.** `/usr/include` et `/usr/local/include`).
- Si `#include <fichier.h>` cherche le fichier les répertoires du système.

Résultat de la commande (appel au pré-processeur seul) : `gcc -E main-sqr.c`

```
# 1 "main-sqr.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 367 "<built-in>" 3
# 1 "<command␣line>" 1
# 1 "<built-in>" 2
# 1 "main-sqr.c" 2
# 1 "./sqr.h" 1

double sqr (double x);
# 2 "main-sqr.c" 2

int main () {
```

Le préprocesseur supprime tous les commentaires, fait les inclusions et autres traitements.

Les lignes de sorties ont la forme
`# numeroligne fichier drapeau`
avec comme valeurs principales du drapeau : 1 début de fichier, 2 retour au fichier, 3 le texte vient d'un fichier en-tête.

Note : `#include <stdio.h>` a été ignorée.

Compilation séparée et bibliothèques logicielles

Pour faciliter la réutilisation de fonctions, au lieu de dsitribuer le code source (fichiers .c), une bibliothèque logicielle (collection de .o) peut être distribuée :

- Bibliothèque **statique**, par exemple `libin103.a`,
  bibliothèque concaténée au fichier exécutable lors de la compilation.
- Bibliothèque **dynamique**, par exemple `libin103.so`,
  bibliothèque est chargée à l'exécution du programme.

Remarque : Un ou plusieurs fichiers .h peuvent être associés à la bibliothèque pour exposer les fonctions accessibles.

Exemple pour la bibliothèque **libin103**

```
bitree.h bitreealg.h bistree.h graph.h list.h stack.h
heap.h queue.h uf.h set.h
```

Exemple d'utilisation de bibliothèque logicielle

Exemple : arborescence d'un projet (avant compilation)

```
Makefile myinclude/ lib1/ lib2/ main.c

./myinclude/: bonjour.h

./lib1/: Makefile bonjour1.c

./lib2/: Makefile bonjour2.c
```

Fichier `main.c`

```c
#include <stdlib.h>

#include "bonjour.h"

int main (int argc, char** argv) {

  bonjour("Alex");

  return EXIT_SUCCESS;
}
```

Fichier `include/bonjour.h`

```c
void bonjour (char* str);
```

Exemple d'utilisation de bibliothèque logicielle

Exemple : arborescence d'un projet (avant compilation)

```
Makefile myinclude/ lib1/ lib2/ main.c

./myinclude/: bonjour.h

./lib1/: Makefile bonjour1.c

./lib2/: Makefile bonjour2.c
```

Fichier `lib1/bonjour1.c`

```c
#include <stdio.h>

void bonjour (char* name) {
  printf ("Salut␣%s\n", name);
}
```

Fichier `lib2/bonjour2.c`

```c
#include <stdio.h>

void bonjour (char* name) {
  printf ("Hello␣%s\n", name);
}
```

Exemple d'utilisation de bibliothèque logicielle

- Règle (simplifiée) de compilation pour `main-static1.x`

```
gcc -Iinclude -o main-static1.x main.c ./lib1/libbonjour.a
```

- Règle (simplifiée) de compilation pour `main-dynamic.x`

```
gcc -Iinclude -o main-dynamic.x main.c -Llib2 -lbonjour
```

Nouvelles options de `gcc`

- `-I` définit un répertoire dans lequel trouver des fichiers .h
- `-L` définit un répertoire dans lequel trouver des bibliothèques .a ou .so
- `-lbonjour` drapeau indiquant l'utilisation de la bibliothèques `libbonjour.{a,so}`
  Note : par défaut, `gcc` lie les bibliothèques dynamiques.

Exemple d'utilisation de bibliothèque logicielle

Exemple : arborescence d'un projet (après compilation)

```
Makefile myinclude/ lib1/ lib2/ main-dynamic.x main-static1.x
  main-static2.x main.c

myinclude/: bonjour.h

lib1/: Makefile bonjour1.c bonjour1.o libbonjour.a libbonjour.so

lib2/: Makefile bonjour2.c bonjour2.o libbonjour.a libbonjour.so
```

```
% ./main-static1.x
Salut Alex
% ./main-static2.x
Hello Alex
% ./code/librairies/main-dynamic.x
ld: Library not loaded: libbonjour.so
  Referenced from: ./main-dynamic.x
  Reason: image not found
% LD_LIBRARY_PATH=./lib1 ./main-dynamic.x
Salut Alex
% LD_LIBRARY_PATH=./lib2 ./main-dynamic.x
Hello Alex
```

Les Makefiles

Makefile

Constatation

Les lignes de compilation peuvent être longues et complexes. La compilation dans ce cas compile tous les fichiers concernés à chaque fois.

Exemple

```
gcc -Wall -Werror -I../include -L../lib -o my_prog.x my_prog.c algo.c -lm -lin103
```

Les fichiers Makefile permettent

- de sauvegarder les règles de compilation ;
- de gérer les dépendances (cf plus loin) pour limiter l'effort de compilation.

Dans ce cours nous allons utiliser l'outil **GNU Make** mais d'autres outils existent (par exemple, **CMake** qui s'appuie sur GNU Make).

Anatomie d'un Makefile

Convention: Un fichier Makefile sera nommé `Makefile`.

Un Makefile contient

- des variables ;
- des règles (ou *recipes* dans la terminologie consacrée).

```
CC=gcc

hello.x: hello.c
	$(CC) -Wall -Werror -o hello.x hello.c
```

Makefile – variables

- Une variable n'est qu'une définition dans un Makefile
- Pour récupérer son contenu on utilise la notation `$()`.

Exemple – variable

Définition d'une variable `cc` donnant la commande pour compiler un fichier .c

```
CC=gcc
```

Exemple – utilisation de variable

Utlisation de la variable CC

```
$(CC) -Wall -Werror -o hello.x hello.c
```

Convention: les variables d'un Makefile sont placées en début de fichier avec des noms en majuscules.

Makefile – règles

Les règles de production ont la forme générale suivante

```
cible: dependance1 dependance2 ...
	actions
```

- `cible` : souvent le nom d'un fichier qui doit être le résultat des actions mais pas que.
- `dépendances` : éléments (séparés par un espace) qui sont nécessaires pour construire la cible
  - elles peuvent être des cibles (intermédiaires) qui doivent être créées avant l'exécutions des actions de la règle courante ;
  - elles peuvent être des noms de fichiers, ils doivent être présents.
- `actions` (recipes) : commandes `shell` pour construire la cible.

Point de vigilance

Vous DEVEZ utiliser une tabulation (touche ⇥) pour indenter vos actions ! !

Makefile – exécution

Pour exécuter les règles d'un Makefile

- Il faut utiliser la commande `make` dans le répertoire contenant le fichier `Makefile` ;
- Par défaut, la commande `make` (sans argument) exécute la première règle dans le fichier `Makefile` ;
- Pour exécuter une règle spécifique, il faut utiliser la commande

```
make target
```

Fonctionnement :

- Si une dépendance doit être reconstruite (issue d'une règle) ou avec une date de modification plus récente que la cible (pour les fichiers), la règle doit être exécutée.
- Une cible qui n'est pas un fichier rend la règle *PHONY*. Cette règle est exécutée inconditionnellement à chaque appel.

Makefile – autre exemple

Un ficher Makefile permet de sauvegarder les règles de compilation mais pas que :

Exemple plus complexe

```
CC=gcc

hello.x: hello.c
	$(CC) -Wall -Werror -o hello.x hello.c

clean: # PHONY recipe
	rm -f hello.o hello.x

distrib: $(USER)-livraison.tgz # PHONY recipe

$(USER)-livraison.tgz: hello.c
	mkdir $(USER)-livraison
	cp hello.c $(USER)-livraison
	tar czf $(USER)-livraison.tgz $(USER)-livraison
	rm -rf $(USER)-livraison

.PHONY: distrib clean # To avoid conflict with rep. with these names
```

Retour sur les pointeurs

Alias de type

Rappel

La commande `typedef` permet de créer des alias de types. Elle a la forme suivante

```
typedef <type existant> <nom>
```

- `typedef unsigned long int ulint;`
- `typedef struct point point_t;`
- `typedef double* vector_t;`
- Forme imbriquée (définition de type et création d'alias)

  ```
  typedef struct point_ { double x; double y; } point_t;
  ```

Point de vigilance

Cette commande ne définit pas un nouveau type mais juste un nouveau nom pour un type connu.

Transtypage (ou cast)

Transtypage

- Pour les types de bases : opération de conversion d'une valeur d'un type en une autre valeur d'un autre type.
- Pour les autres types : opération de changement de type d'une variable prise en compte à la compilation.

```
int x = (int) 3.14159; /* x = 3 */
```

Points de vigilance

- Possibilité de perte / corruption d'information ;
- La conversion entre entier et flottant change la représentation interne.
- Pour les pointeurs cela change uniquement la façon d'interpréter la zone mémoire associée.

Généricité en C

Un pointeur est une adresse correspondant à une zone mémoire dont la taille dépend du nombre et du type d'information qui y est stockée ⇒ pas possible d'affecter des pointeurs de type différents.

Le programme

```
#include <stdlib.h>

int main(int argc, char **argv)
{
  int* a = NULL;
  double *b = NULL;
  a = b;

  return 0;
}
```

génère une erreur à la compilation

```
toto.c:7:5: error: incompatible pointer types assigning to 'int␣*'from 'double␣*'
```

Généricité en C

- Dans différentes situations, il peut être utile de manipuler (stocker) des pointeurs sans connaître exactement les types de valeurs qu'ils référencent.
- Pour cela on utilise des pointeurs génériques `void*`

```c
int a = 10;
char b = 'c';

/* sans erreur */
void *p = &a;

/* sans erreur */
p = &b;
```

```c
#include <stdio.h>

int main()
{
    int a = 10;
    void *ptr = &a;
    printf("%d", *((int *)ptr));
    return 0;
}
```

Point de vigilance

Pour utiliser un pointeur générique, il faut le convertir dans le type de données adéquate $\Rightarrow$ il faut mémoriser (dans le code ou dans sa tête) ce qu'il représente.

Exemple

```
void swapg (void* a, void* b, int size) {
  void *tmp = malloc (size);
  /* void* memcpy(void *dst, void *src, size_t n); */
  memcpy (tmp, a, size); /* copie une zone memoire */
  memcpy (a, b, size);
  memcpy (b, tmp, size);
  free (tmp);
}

int main(int argc, char **argv)
{
  int a = 1; int b = 4;
  printf ("a = %d, b = %d\n", a, b);
  swapg((void*)&a, (void*)&b, sizeof(int)); /* param. int */
  printf ("a = %d, b = %d\n", a, b);

  double c = 1.1; double d = 4.1;
  printf ("c = %f, d = %f\n", c, d);
  swapg((void*)&c, (void*)&d, sizeof(double)); /* param. double */
  printf ("c = %f, d = %f\n", c, d);

  return 0;
}
```