# Projet de projet(s)

11 Février 2019 — Jawher Jarrey, Hiba Ouni, Pierre Rousselin et Xavier Monnin

Voici deux sujets de projet assez différents. Les consignes sont les suivantes :

— Seul un des deux sujets est à traiter.

— Vous ferez ce travail en binôme (de préférence) ou seul si ce n'est pas possible.

— Vous devrez faire preuve d'au moins un peu d'imagination et aller (un peu) au delà de ce qui est mentionné dans le sujet.

— Tout échange de code entre des binômes différents est interdit.

— Tout copier-coller depuis des codes lus sur le web ou dans des livres est interdit.

## 1 Le programme `autoprog2`

Ce sujet de projet fait référence au sujet de TP n°2 du cours « Programmation 2 ». Pour rappel, on définissait dans ce sujet plusieurs types structurés (point, cercle, rectangle, roue, voiture) « simples » (sans champ tableau ni champ pointeur) et pour chaque type `A` on demandait de déclarer et définir les fonctions :

— `A creer_A(type1 champ1, ..., typen champn);` qui initialise un objet de type `A` à partir des valeurs de ses différents champs ;

— `void afficher_A(A x);` qui affiche les champs de l'objet `x` de type `A` ;

— `int comparer_A(A x, A y)` qui retourne $-1$ si `x` est plus petit que `y`, 0 s'ils sont égaux et `1` si `x` est plus grand que `y` ;

— `void echanger_A(A *x, A *y)` qui échange les valeurs pointées par `x` et `y`.

Parmi ces fonctions, la première et la dernière peuvent être écrites de façon entièrement automatique. Pour la deuxième et la troisième, on peut en donner automatiquement certains morceaux (au moins la déclaration).

Votre travail sera tout d'abord, d'écrire un script shell qui, étant donné un fichier en entrée (ou depuis l'entrée standard) qui ressemblerait *par exemple* à :

```
struct point_s {
  double x;
  double y;
};
struct cercle_s {
  point centre;
  double rayon;
};
/* etc */
```

crée :

— Pour chaque type structuré `A` un fichier `A.h` contenant :

  — un garde d'inclusion ;

  — la définition de la structure `struct A_s` ;

  - l'alias de type `typedef struct A_s A;`
  - la déclaration de chacune des quatre fonctions ci-dessus ;
  - les directives `#include` nécessaires à ce fichier d'en-tête.
- Pour chaque type structuré `A` un fichier `A.c` contenant :
  - les directives `#include` nécessaires ;
  - les définitions, quand c'est possible, des fonctions ci-dessus, sinon des déclarations supplémentaires avec la mention `/*TODO*/` en commentaire.
- Un fichier `main.c` (réduit à un affichage de la chaîne `"Hello, World!"`) utile pour tester la compilation ;
- Un fichier `Makefile` complet similaire à celui vu en TP de programmation 2.

La compilation avec `make` ne doit produire ni erreur ni avertissement. Ceci est le minimum. Pour aller plus loin, voici quelques idées (mais vous êtes encouragés à avoir les vôtres !) :

- La syntaxe du fichier en entrée présenté plus haut est assez rigide. Vous pouvez essayer de voir si vous pouvez traiter d'avantage de syntaxes différentes. Par exemple :

```
struct point_s
{
      double x, y;
};
```

- Vous pouvez essayer de détecter certaines erreurs de syntaxe dans ce fichier d'entrée.
- Vous pouvez essayer de détecter des champs qui seraient des pointeurs ou des tableaux dans le fichier d'entrée.
- Vous pouvez essayer de traiter le cas de champs qui seraient des tableaux statiques ou des chaînes de caractère.
- Si tous les champs sont formés de types « de base » (chose que l'on peut tester) on peut écrire entièrement la fonction d'affichage.
- On peut ajouter la déclaration et la définition d'une fonction qui trie (par la méthode de votre choix) un tableau d'éléments de type `A` par ordre croissant (où bien sûr l'ordre est déterminé par `comparer_A`).
- On peut ajouter la déclaration et la définition d'une recherche (efficace) d'un élément de type `A` dans un tableau trié d'éléments de type `A`.
- ...

## 2 Un gestionnaire de « *To-do list* »

Une *To-do list* est simplement une liste, plus ou moins détaillée de choses à faire. Ce projet consiste à implémenter un programme en shell nommé `todo` qui sert, au niveau le plus basique, à l'utilisateur à :

- écrire des nouvelles entrées dans sa *To-do list* ;
- afficher sa *To-do list* ;
- rayer des entrées dans sa *To-do list*.

Voici un exemple de déroulement basique (le nom des options peut être changé, l'affichage aussi, **ce n'est qu'un exemple de ce qui est possible**) :

```
$ todo "Finir le projet de shell"
$ todo "Réviser le C pour le partiel"
$ todo --list
Réviser le C pour le partiel -- ajouté le 11/02/2019 à 15h02
Finir le projet de shell -- ajouté le 11/02/2019 à 14h45
Acheter des couches -- ajouté le 10/02/2019 à 22h13
Faire du jogging -- ajouté le 1/01/2019 à 00h01
$ todo --done "Acheter des couches"
$ todo --list
Réviser le C pour le partiel -- ajouté le 11/02/2019 à 15h02
Finir le projet de shell -- ajouté le 11/02/2019 à 14h45
Faire du jogging -- ajouté le 1/01/2019 à 00h01
```

Là aussi, ce n'est que le strict minimum. On pourra implémenter beaucoup de choses supplémentaires. Quelques suggestions :

— Pouvoir faire référence à une tâche avec seulement une partie du texte, lister les tâches contenant une chaîne, une expression rationnelle.

— Donner une priorité à certaines tâches et pouvoir trier par priorité décroissante la liste.

— Donner pour certaines tâches une date limite et pouvoir afficher d'abord les tâches dont la date limite est la plus proche.

— Afficher une autre liste qui fait du bien : tout ce qui a déjà été fait.

— Ajouter des commentaires contenant des informations sur une tâche.

— Gérer plusieurs listes (par exemple une liste `courses` en plus de la liste par défaut).

— Ajouter un fichier de configuration (par exemple `.todorc`) dans le répertoire personnel avec des réglages (nom du ou des fichiers contenant les listes, etc) qui seraient lus au démarrage du programme.

— Une *To-do list* peut être quelque chose de très privé. On pourrait imaginer crypter et décrypter le fichier en utilisant un mot de passe.

— ...