TD : Programmation en C — semaine 7

29 mars 2018

Objectif(s)

- ✯ Modularité dans les programmes C.
- ✯ Introduction à la structure de liste chaînée.
- ✯ Renforcement de la notion de pointeurs.

— **Note 1** : Pour tous vos programmes, vous utiliserez la commande `make` pour compiler.
— **Note 2** : Pour tous vos programmes, vous ferez des fonctions de 5 lignes maximum (sans comptez les commentaires, accolades seules ou en-têtes de fonctions).
— **Note 3** : Pour tous vos programmes, vous ferez des fichiers de 50 lignes maximum (sans comptez les commentaires).

Exercice 1 – Diviser en sous-fichiers

Avoir un programme dans un unique fichier est très peu lisible quand le projet devient conséquent. Il est beaucoup plus propre de diviser son projet en différents fichiers contenant des concepts distincts :
— Un fichier responsable d'afficher le menu du programme,
— Un fichier contenant des algorithmes pour trier un tableau,
— Un fichier permettant de saisir des entrées utilisateurs (chaines de caractères, entiers, ...),
— ...
Le plus grand avantage d'effectuer cette division est de pouvoir ré-utiliser des fichiers au travers de différents projets. Par exemple, un fichier `saisir.c` peut proposer des fonctions pour saisir des entrées utilisateurs. Il est facile de comprendre ce que fait ce fichier dans un programme : diviser des concepts en sous-fichiers permet de réduire la difficulté d'appréhender un programme.

Un deuxième point très important et intrinsèque à la division en petit concepts est qu'un fichier permet d'isoler l'implémentation d'un traitement. Par exemple, pour utiliser la fonction `saisir_chaine`, on a juste besoin de savoir qu'elle prend une chaine en entrée, une taille maximale et qu'elle ne renvoie rien ; ce qui se traduit avec *le prototype* (ou signature) de la fonction suivant :

```
void saisir_chaine(char* s, int size_max);
```

Comme on peut le constater, l'implémentation de la fonction est "effacé". En effet, on a pas besoin de savoir que la fonction utilise `fgets` pour l'utiliser. Pour distinger l'implémentation de la fonction de sa signature, on utilise deux fichiers distincts : `saisir.h` et `saisir.c`. Le `.h` contient le prototype de la fonction et le `.c` son implémentation, par exemple :

Listing 1 – "saisir.h"

```
#ifndef _SAISIR
#define _SAISIR

void saisir_chaine(char* s, int size_max);

#endif
```

Listing 2 – "saisir.c"

```c
#include "saisir.h"

void saisir_chaine(char* s, int size_max) {
  fgets(s, size_max, stdin);
  int size = strlen(s);
  // ...
}
```

Tout ce qui commence par # fait partie de ce qu'on appelle le *pré-processeur*. C'est très simple : le pré-processeur est un méchanisme de copier-coller automatique, par exemple :

— Si on a `#define N 5`, on va remplacer tous les symboles N dans notre fichier par la valeur 5, et donc le nom N disparait (c'est pour ça qu'on ne peut pas faire `N = N + 1;` qui est remplacé par `5 = 5 + 1;`).

— C'est la même chose avec `#include "saisir.h"`, le fichier `.h` est copier-coller dans le fichier `.c`.

1. Créer un projet dans un dossier dédié avec trois fichiers :
   — `saisir.h` et `saisir.c` pour la saisie utilisateur.
   — `main.c` qui contiendra le main est demandera à l'utilisateur son nom et l'affichera (en utilisant les fonctions de `saisir_chaine`).
   Compiler le projet n'est plus aussi simple qu'avant puisqu'on doit compiler tous les fichiers `.c`—pas les `.h` vu qu'ils sont copier/coller dans les `.c`. Le problème est que `run.sh` ne marche que pour un seul fichier, on va utiliser un outil plus puissant qui s'appelle `Makefile`. Vous pouvez récupérer ce fichier, déjà codé pour vous, sur le site du cours.
2. Modifier les champs `SRC` dans le fichier `Makefile` pour ajouter votre fichier `saisir.c`. Vous pouvez essayez de compiler en lançant `make` (au lieu de `run.sh`). Le nom du programme à exécuter est contenu dans le champs `EXEC` du fichier `Makefile`.
3. En partant du jeu de la bataille, divisez votre projet en plusieurs fichiers tel que chaque fichier contient un concept distinct.

Exercice 2 – Déchainez-vous !

On va voir une nouvelle structure de données : les listes chainées. Abstraitement, elles permettent de programmer des tableaux avec l'avantage qu'on puisse facilement enlever ou ajouter des éléments à n'importe quelle position de la liste. Notez qu'un algorithme qui ajoute ou enlève un élément dans un tableau demande de décaler tous les éléments à droite de cette position.

Le type d'une liste chainée d'entiers est comme suit :

```c
struct Noeud {
  struct Noeud* suivant;
  int valeur;
}
```

On a une structure principale `Noeud` qui contient une valeur (`noeud.valeur`) et l'adresse du nœud suivant (`noeud.suivant`).

1. Implémentez une fonction qui permet d'ajouter un nouveau noeud dans une liste :

   ```c
   Noeud* ajout_debut(Noeud* liste, int valeur);
   ```

   L'astuce pour créer le premier élément est d'utiliser `Noeud* liste = ajout_debut(NULL, 34)` pour créer une liste d'un seul élément avec la valeur 34.
2. Mettez les fonctions et prototypes relatifs à la liste chainée dans un fichier `liste.c` et `liste.h`. Vous ferez des tests dans un fichier `main.c`. Modifier le `Makefile` pour compiler.
3. Créer une fonction qui affiche la liste :

   ```c
   void affiche_liste(Noeud* liste);
   ```

Vérifiez que votre fonction `ajout_debut` marche bien. Implémentez une version itérative (avec boucle `while`) et une version récursive.

4. Créer une fonction qui libère la mémoire occupée par la liste (en utilisant `free` sur chacun des nœuds) :

```
void libere_liste(Noeud* liste);
```

5. Ajouter les fonctions suivantes :

```
Noeud* ajout_fin(Noeud* liste, int valeur);
int taille(Noeud* liste);
// Enlever le noeud a la position "indice" et retourne la liste sans le noeud.
Noeud* enlever_noeud(Noeud* liste, int indice);
```

6. Implémentez les versions récursives et itératives des fonctions précédentes.