Programmation C avancée
TP 8

L'objectif de ce TP est d'élaborer un mécanisme automatique de branchement de plugins sur un programme relativement simple et déjà connu : dc (Desk Calculator). Même s'il n'est pas difficile de faire fonctionner un prototype, on tâchera de gérer correctement les erreurs et contrôles de plages de valeurs.

Encore une fois, il faudra lire une grande quantité de documentation ainsi que les supports de cours pour espérer obtenir un code qui fonctionne correctement.

Souvenez-vous de la calculatrice en Polonais inversé du TP3. Il y avait une pile, des entrées ligne à ligne pour les commandes utilisateurs et des opérations qui dépilaient des arguments pour ensuite empiler leurs résultats. Le but est maintenant de transformer les opérations en plugins dont le chargement sera COMPLÈTEMENT automatique à l'exécution du programme.

Ce mécanisme est largement utilisé par exemple dans les jeux vidéos pour lesquels suivant le niveau de difficulté sélectionné par le joueur, une librairie dynamique plus ou moins sophistiquée est chargée (Toutes ces bibliothèques d'intelligences artificielles contiennent les mêmes symboles pour le linker et donc respectent des spécifications TRÈS TRÈS précises).

Plan d'attaque

Voici un petit résumé des opérations (dont on peut librement s'écarter si l'on considère avoir suffisamment compris la problématique) :

- reprendre le code du TP3 à propos de la calculette en Polonais inversée
  - garder les entrées/sorties (pas de raisons de changer)
  - faire une bibliothèque statique avec le module à propos de la gestion de pile
  - nettoyer le code à propos des opérations arithmétiques, on va le refondre
- créer un répertoire `plugins` dans votre TP
  - créer un fichier `.c` par opération arithmétique
  - enrichir votre `Makefile` en conséquence, chacun des plugins doit être compilé de manière à produite une bibliothèque dynamique (`-fPIC` et `-shared` seront nécessaires).
- recherche des plugins disponibles pour votre programme
  - charger la bibliothèque `<dirent.h>`
  - faire une fonction de filtrage (comme suggère la documentation) pour `scandir` (le filtrage peut consister à ne sélectionner que les `.so` du répertoire `plugins`).
  - appeler `scandir` avec votre fonction de filtrage et `alphasort` (bien placer la macro `_SVID_SOURCE` avant l'inclusion de `dirent.h` pour éviter les warnings); on veillera à bien noter la valeur de retour qui représente le nombre de plugins à charger

- chargement des plugins
  - créer une jolie structure pour les plugins et allouer un tableau de cette structure dont la longueur est le nombre de plugins à charger
  - charger la bibliothèque <dlfcn.h>
  - pour chaque plugin `.so` trouvé, ouvrir la bibliothèque avec `dlopen` et rechercher les symboles que l'on stockera à l'endroit adéquat (dans le tableau de plugins)
- utilisation des plugins
  - implanter une fonction `int apply_operation(char symb)` qui prend en argument un symbole d'opération. Votre fonction devra dépiler autant d'entiers que l'arité correspondante, les donner en arguments de la fonction d'évaluation correspondante, récolter le résultat et l'empiler sue la pile de votre programme.
  - greffer la fonction précédente dans vos anciennes entrées sorties pour appliquer l'opération si possible, afficher un message sur `stderr` si le symbole ne correspond à aucun plugin chargé

Spécifications pour les plugins

Vos plugins devront OBLIGATOIREMENT COMPORTER ces trois symboles suivants:

- une fonction d'évaluation de type `int eval(int* args)`.
- une fonction `int arity(void)` sans argument retournant le nombre d'argument de la fonction d'évaluation.
- une fonction `char symbol(void)` sans argument retournant le caractère ASCII représentant l'opération.

La nature du problème veut ici que les plugins sont tout petit et très simples à écrire. Ce n'est le cas dans la réalité mais la difficulté de ce TP se situe ailleurs...

Attention, les noms (chaînes de caractères ASCII) sont très importants. Sinon les tentatives de récupération des symboles échoueront (`eval`, `arity`, `symbol`).

Précisions pour `scandir`

La documentation est toujours obtenue avec `man scandir`. Une macro doit être définie avant l'include des headers de la bibliothèque pour une bonne résolution des liens par `gcc`.

Voici des précisions sur la structure `dirent`:

```
1 struct dirent
2 {
3   long d_ino;                 /* inode number */
4   off_t d_off;                /* offset to this dirent */
5   unsigned short d_reclen;    /* length of this d_name */
6   char d_name[NAME_MAX+1];    /* file name (null-terminated) */
7 };
```

Seul le champ `d_name` nous intéressera. Ce dernier contient le nom du fichier attrapé par `scandir`. La fonction de filtrage teste si ce nom est d'extension `.so` auquel cas, elle retourne 1. Sinon votre fonction de filtrage devra retourner 0 et donc se libérer de la cible croisée.

Pour la libération mémoire, il faut traiter un `struct dirent**` comme tout tableau à deux dimension classique. On libère les lignes (les `struct dirent*`) puis le tableau principal qui contient l'adresse de chaque ligne (le `struct dirent**`).

Précisions pour `dlopen`

Avant d'ouvrir nos bibliothèques dynamiques, on va devoir concevoir une variable (avec son type) pour stocker et manipuler de manière sympathique nos plugins. Voici une structure simple qui respecte les spécifications de nos plugins :

```
1 typedef int (*eval)(int* args);
2
3 typedef struct{
4   eval eval;
5   int arity;
6   char symbol;
7 }Plug_symbol;
```

Suivant le retour de `scandir`, on peut borner le nombre maximum de plugins à charger (maximum car rien ne nous garantie que tout les `dlsym` trouveront les symboles à priori). Ayant un nombre max de plugins, il nous faudra allouer un tableau de `Plug_symbol` de taille égale pour stocker les résultats des `dlsym` à venir.

Dans un objectif de sécurité, vérifier tous les résultats des fonctions issues de la bibliothèque <dlfcn.h> (sauf pour `dlclose`). Cela consiste à vérifier que `dlerror` pointe toujours sur `NULL` après toute action sur les bibliothèques dynamiques.

Tests finaux

Ce TP offre un test très simple de contrôle pour vérifier que votre implémentation satisfait le cahier des charges décrit plus haut. Effacer tout vos fichiers `.so` de votre répertoire `plugins`. Prenez via clé usb ou autre les fichiers plugins `.so` d'un de vos camarades partageant la même architecture que vous (Linux 32 bits i384 ou bien Linux 64 bits amd64). Copier ces fichiers dans votre répertoire `plugins` et sans rappelez votre `Makefile`, relancer votre programme. Vous devriez alors jouir des opérations arithmétiques implantées par votre camarade en plug and play. Avec une telle implémentation, l'installation de nouveaux plugins se réduit donc au copier-coller.

Voici une suggestion de `verbose` pour votre exécutable :

```
nborie@perceval:~> ls plugins
addition.so  multiplication.so   negative.so
nborie@perceval:~> ./eap
Auto-loading of plugins...
found 3 potential plugins :
- addition.so... loading OK
- multiplication.so... loading OK
- negative.so... loading OK
...
```