TP n°15 - Programmation modulaire - Librairies en C.

On rappelle que pour pouvoir exécuter les programmes compilés sur les machines du lycée, votre répertoire de travail doit être situé en dehors du répertoire `Documents`

Pour le moment, tous les codes que vous avez réalisés tenaient dans un seul fichier source : toutes les fonctions, ainsi que la fonction principale, étaient écrites dans un seul fichier.

Bien sûr, cette méthode est pratique mais elle a ses limites. Certains codes industriels font plusieurs millions de lignes de code... imaginez s'il fallait travailler dans un seul fichier ! Cela poserait d'énormes problèmes en terme de travail collaboratif, de modularité...

Nous allons revoir dans ce premier exercice la chaîne de compilation et notamment l'étape d'édition de liens, qui permet de relier des codes binaires issus de la compilation de différents fichiers.

Exercice 1 (Compilation séparée et script shell).

1. Sur papier, rappeler les étapes de la chaîne de compilation et rappeler les différentes commandes permettant d'arrêter le processus sur l'une des étapes.
2. Dans votre répertoire du TP15, créez, en utilisant le shell, un sous-répertoire nommé `compilation-separee`
3. Téléchargez les fichiers `fichier1.c` et `fichier2.c` disponibles sur Moodle et déplacez les vers le répertoire que vous venez de créer à l'aide d'une commande shell
4. Ouvrez les deux fichiers et observez leur contenu
5. Créer le fichier objet, que vous nommerez `fichier1.o`, associé au fichier `fichier1.c`
6. Analysez sa table des symboles avec l'outil `readelf -s` suivi du nom du fichier objet, et/ou avec l'outil `objdump` suivi de l'option `-t`
7. Faites de même avec le fichier `fichier2.c`
8. Créer un programme exécutable en réalisant l'étape d'édition de liens entre les deux codes
9. Rassemblez toutes les opérations effectuées dans un script shell et vérifiez son bon fonctionnement
10. **En faisant très attention**, car vous pourriez effacer tout le contenu de votre répertoire et même plus, rajouter des lignes permettant de nettoyer le répertoire courant. Commencez par essayer votre commande dans un sous répertoire temporaire pour être sûr de vous.

En réalité, pour automatiser les tâches de compilation sur de gros codes constitués d'un grand nombre de fichiers, on utilise rarement des scripts shell. Il existe un utilitaire dédié, appelé ***`makefile`***

Dans tous les langages de programmation dits de haut niveau, il existe des librairies : elles regroupent des fonctions déjà implémentées par des codeurs expérimentés, réutilisables par d'autres programmeurs. Les librairies permettent de gagner du temps en bénéficiant du travail d'autres personnes ayant déjà implémenté et testé la librairie, et la mettant à jour régulièrement.

Remarque. Nous considérerons que les expressions suivantes sont synonymes :

— modules (plutôt en OCaml),
— packages (plutôt en Python),
— librairies (plutôt en C),
— bibliothèques de fonctions (le vrai nom en français)

Nous avons appris comment utiliser des librairies en C. Il vous a suffi d'utiliser la directive de prétraitement `#include` suivi du nom d'un fichier d'en-tête (extension `.h`) pour bénéficier de toutes les fonctions de

différentes librairies.

Exercice 2 (Bibliothèques en C).

1. Rappeler l'entête et l'usage des fonctions `strcpy` et `strcmp`. Dans quelle librairie se trouvent-elles ?
2. Les fichiers d'en-tête des librairies installées sont stockées sur le disque dur, dans la partie du système de fichiers réservée au système d'exploitation. Sur votre système Linux, ils se trouvent généralement dans le dossier système `/usr/include`. Ouvrir dans `emacs` le fichier d'en-tête de la librairie `string` et l'analyser. Que contient-il ?
3. Aller dans le répertoire système `/usr/lib32` ou sinon `/lib/x86_64_linux_gnu` et lister tous les éléments contenus dans ce répertoire. Ce sont des fichiers qui rassemblent les codes des librairies sous forme compilée. Choisissez quelques librairies présentes dans ce répertoire et recherchez sur Internet les fonctionnalités qu'elles proposent. Avez-vous trouvé dans quel fichier se situe le code compilé associé aux fonctions `strcpy` et `strcmp` ?
4. Utiliser la commande `readelf` pour avoir aux accès à la table des symboles de la `libc`.
   Afficher la table des symboles contenus dans cette librairie et vérifier la présence des fonctions `strcpy` et `strcmp`.

Une librairie C est donc constituée de deux parties :

Une interface, sous la forme d'un fichier d'extension `.h` contenant uniquement les structures de données définies dans la librairie et les prototypes des fonctions implémentées par la librairie. L'interface ne contient aucun code, aucune implémentation technique. Elle décrit juste les nouveaux types d'objets définis par la librairie s'il y en a, et les fonctions qui pourront être utilisées par les clients de la librairie (les programmeurs utilisant la librairie).

Une implémentation, qui contient les instructions binaires associées au code source des fonctions de la librairies. Le code d'une librairie ne contient pas de fonction principale `main`. Il contient uniquement des fonctions qui seront utilisées dans un autre code, appelé **code client** qui, lui, contiendra bien sûr une fonction `main`.

Concernant l'implémentation, il existe des librairies

statiques : sous Linux, un fichier de librairie statique porte l'extension `.a`. Une librairie statique est reliée au code client utilisant la librairie au moment de la compilation. Si la librairie change, le code de la personne utilisant la librairie doit être recompilé.

dynamiques : sous Linux, un fichier de librairie dynamique porte l'extension `.so`. Une librairie dynamique est reliée au code client utilisant la librairie au moment de chaque exécution, lors du chargement du code en mémoire vive. Cela prend un peu de temps et limite l'optimisation du code, mais est plus modulaire : si la librairie change et reste conforme à l'interface, alors le code client n'a pas besoin d'être recompilé. La nouvelle librairie sera automatiquement *linkée* lors du chargement (*load*) du programme binaire en mémoire de travail, juste avant l'exécution.

Exercice 3 (Création d'un librairie dynamique).
Le but de cet exercice est de transformer notre code sur les listes chaînées en une librairie dynamique potentiellement utilisable par un autre programmeur. Pour cela, nous allons remodeler le code et le compiler d'une manière particulière pour être en mesure de mettre à disposition de notre client utilisateur une interface sous la forme d'un fichier d'en-tête, et l'implémentation concrète sous la forme de fichiers objets (au format ELF).

1. Création d'une librairie dynamique

a. Reprendre le code `listes_maillons_ex2.c` du TP12. On pourra utiliser le code corrigé déposé sur Moodle.

b. Remodeler ce code `liste-maillons.c` en deux fichiers : un fichier d'en-tête `mylist.h` qui servira d'interface, et un fichier `mylist.c` qui contiendra l'implémentation concrète des fonctionnalités de notre bibliothèque. Pour le fichier d'interface, on prendra soin de rajouter le mot clé `extern`, nécessaire en C, devant le prototype de chaque fonction.

c. Créer le **fichier objet** associé aux instructions codées dans `mylist.c`. Vous pouvez retourner voir le cours de la Séquence 2 sur la chaîne de compilation. *On ne doit pas aller jusqu'au bout de la chaîne de compilation.* Attention, il faudra utiliser l'option `-fPIC` lors de la création du fichier objet pour générer du code déplaçable. Je vous explique à l'oral ce que cela signifie.

d. Pour convertir le fichier objet `mylist.o` en une librairie dynamique `libmylist.so`, taper la ligne de commande suivante dans le Terminale :

```
gcc -o libmylist.so -shared mylist.o
```

On obtient alors un fichier de librairie dynamique `libmylist.so`.

La commande `readelf -s libmylist.so` permet d'obtenir la table des symboles de la librairie dynamique, en particulier la liste des noms de fonctions qui apparaissent dans le code binaire et qui pourront donc être reliées à un autre code client utilisant cette librairie. Observez la table des symboles de la librairie dynamique `libmylist.so` nouvellement créée.

2. Utilisation d'une librairie dynamique

a. Créer un code `test_lib.c` qui utilise la bibliothèque nouvellement créée comme vous en avez déjà l'habitude avec les librairies systèmes que nous utilisons régulièrement. On pourra faire des appels aux fonctions de la bibliothèque dans le code (soit dans le `main`, soit dans d'autres sous-fonctions).

b. Compiler le code en rajoutant l'option `-lmylist` sur la ligne de commande (on enlève le préfixe`lib` dans le nom de la librairie après l'option `-l`). Cette option sert :

— à indiquer au compilateur que certains liens ne seront pas résolus immédiatement

— à lui demander d'intégrer dans l'exécutable crée au format ELF une information indiquant que ce code utilise la librairie dynamique `libmylist.so`. Le logiciel de chargement (l*oader*) utilisera cette information à chaque nouvelle exécution du code : il chargera le code binaire de votre librairie en plus de celui de votre code de test et finalisera l'édition de liens.

Si `gcc` ne trouve pas votre librairie dynamique, forcez le à rechercher votre librairie dynamique dans le répertoire courant en rajoutant l'option `-L.` sur la ligne de commande

c. Exécuter le code de test et vérifier son bon fonctionnement

d. Que se passe-t-il ? En fait, par défaut, le chargeur de programme ne sait pas où aller chercher la librairie dynamique que vous venez de créer. Pour lui indiquer d'aller chercher dans le répertoire courant, il faut rajouter ce chemin dans la variable d'environnement dédiée `LD_LIBRARY_PATH` (LD pour LoaDer, pour le chargeur). Faites-le, vérifiez que cette variable a bien été mise à jour et relancez votre code.

3. Refaites toute la chaîne en écrivant toutes les étapes de créations et d'utilisation de librairie dans un script shell. Vous pourrez rajouter des commentaires dans votre script en rajoutant un # en début de ligne, comme en Python.

Exercice 4 (Suite - Mise en évidence du caractère dynamique de l'édition de liens).

1. Sauvegarder la librairie dynamique `libmylist.so` dans un fichier `libmylist-sav.so`
2. Modifier la forme de l'affichage dans la fonction `print_list` du fichier `mylist.c` pour indiquer également la position dans la liste de chacun des éléments affichés
3. Régénérer la librairie dynamique `libmylist.so`
4. Relancer directement votre code de test sans le recompiler. Vous devez constater que c'est bien la nouvelle librairie qui a été chargée.
5. Remplacer `libmylist.so` par son ancienne version qui a été sauvegardée `libmylist-sav.so`
6. Relancer votre code de test, toujours sans le recompiler. Vous devez constater que c'est bien la librairie initiale qui a été chargée.

Exercice 5 (DM vacances).
Vous allez maintenant travailler par groupe de 2 personnes. Tout peut se faire à distance, en communiquant uniquement par mail, c'est même souhaitable !
Le but est de vous faire comprendre l'intérêt d'une interface abstraite en terme de modularité mais aussi de travail collaboratif.
Chaque personne va adopter à la fois le rôle de concepteur et le rôle d'utilisateur client.

Le concepteur est la personne qui va concevoir la librairie.

L'utilisateur-testeur client est la personne qui va utiliser et tester la librairie en ayant à disposition uniquement l'interface fournie par le concepteur et la librairie dynamique au format binaire ELF.

La personne 1 sera conceptrice pour le projet 1 et utilisatrice pour le projet 2. La personne 2 sera conceptrice pour le projet 2 et utilisatrice pour le projet 1.

Le projet 1 consiste à implémenter une bibliothèque permettant de manipuler des fractions. Cette bibliothèque se nommera `myfrac` et devra définir une structure de données nommée `myfrac` permettant de représenter une fraction. Cette bibliothèque exposera 7 fonctions : un constructeur `frac_create`, un destructeur `frac_free`, et 4 fonctions `frac_add`, `frac_sub`, `frac_div` et `frac_mult` prenant à chaque fois 2 fractions en entrée et retournant une fraction, ainsi qu'une fonction `frac_print` permettant d'afficher une fraction.

Le projet 2 consiste à implémenter une bibliothèque permettant de manipuler des nombres complexes. Cette bibliothèque devra définir une structure de données nommée `mycomplex` permettant de représenter un nombre complexe. Cette bibliothèque exposera 7 fonctions : un constructeur `complex_create`, un destructeur `complex_free`, et 4 fonctions `complex_add`, `complex_sub`, `complex_div` et `complex_mult` prenant à chaque fois 2 nombres complexes en entrée et retournant un nombre complexe, ainsi qu'une fonction `complex_print` permettant d'afficher un nombre complexe.

- Chaque concepteur suivra scrupuleusement le cahier des charges décrit ci-dessus pour ces deux structures de données. Il s'attachera notamment à écrire le fichier d'en-tête avant de se lancer dans le code, pour bien délimiter le travail nécessaire pour l'implémentation concrète.
- Chaque concepteur devra fournir l'interface `.h` et le fichier `.so` de son projet à son testeur (avec une clé USB par exemple)
- Le client-testeur devra créer un petit code testant **toutes** les fonctionnalités de la librairie fournie (par exemple dans la fonction principale) et vérifier son bon fonctionnement. Les tests devront être exhaustifs, et envisager toutes les possibilités d'erreurs.
- En cas de bug, il rapporte l'anomalie au concepteur, qui corrigera sa bibliothèque (on appelle cela *patcher*) et lui remettra à disposition le fichier `.so`. L'utilisateur ne doit pas avoir besoin de recompiler son code pour tester la nouvelle version fournie, si l'interface est restée la même.

Si votre gestionnaire de mails n'accepte pas l'envoi de librairie dynamiques par peur des virus, vous pouvez envoyer une archive zippée à votre camarade, qui contient l'interface et la librairie dynamique. Pour cela :

- Créez un dossier `libtp`
- Copiez dans ce dossier votre fichier d'interface et votre librairie dynamique dans ce dossier
- Placez vous dans le dossier contenant le sous-dossier `libtp` si vous n'y êtes pas déjà
- Créez une archive zippée de ce dossier en tapant la commande :

```
tar cvzf ./libtp/* libtp.tgz
```

- Envoyez l'archive zippée `libtp.tgz` à votre camarade
- Le camarade/client enregistre le fichier `libtp.tgz` dans le répertoire dans lequel il va tester votre librairie
- Il désarchive et dézippe ce fichier en tapant :

```
tar xvf libtp.tgz
```

ce qui a pour effet de recréer le sous-répertoire `libtp`, qui contient les deux fichiers, là où il se trouve dans son arborescence

Remarque. Le gestionnaire de paquets `apt` que vous avez utilisé à maintes reprises pour installer des logiciels permet aussi de rechercher des mises à jour de librairies. Il va alors rechercher sur le serveur distant diffusant cette librairie s'il existe une version plus récente (*update*). Si c'est le cas, il télécharge la nouvelle version de la librairie dynamique `.so`.

Si tout va bien, cette mise à jour est totalement transparente pour les codes situés sur votre machine et utilisant cette bibliothèque. Le *loader* ira simplement charger en mémoire la nouvelle version de la librairie lors de la prochaine exécution de ces codes clients. En fait, tant que l'interface de la librairie ne change pas, aucune modification ni re-compilation n'est nécessaire dans le code client.