# TP n°4

## Langage Procédural : C

### Utilisation et création de librairies

L'objectif de ce TP est de manipuler l'éditeur de lien afin *1)* de pouvoir utiliser dans un programme une librairie tierce, et *2)* de créer une nouvelle librairie à partir d'un code que vous avez créé.

## 1 Utilisation d'une librairie

### 1.1 Rappel sur les étapes de compilation

La première étape consiste à écrire le code source en langage C ou C++ (fichiers .c et .h en C et .cpp et .hpp en C++). Ensuite on lance une compilation, par exemple avec gcc (en C) ou g++ (en C++). La compilation (au sens vague du terme) se déroule en trois grandes phases.

**La précompilation (*préprocesseur*)** Le compilateur commence par appliquer chaque instruction passée au préprocesseur (toutes les lignes qui commencent par un `#` , dont les `#define`). Ces instructions sont en fait très simples car elles ne consistent en gros qu'à recopier ou supprimer des sections de codes sans chercher à les compiler. C'est de la substitution de texte, ni plus ni moins.

C'est notamment à ce moment là que les `#define` présents dans un fichier source (.c ou .cpp) ou dans un header (.h ou .hpp) sont remplacés par du code C/C++. À l'issue de cette étape, il n'y a donc plus, pour le compilateur, d'instructions commençant par un `#`.

**La compilation** Ensuite, le compilateur compile chaque fichier source (.c et .cpp), c'est-à-dire qu'il crée un fichier binaire (.o) par fichier source, excepté pour le fichier contenant la fonction main. Cette phase constitue la compilation proprement dite.

Ces deux premières étapes sont assurées par cc lorsqu'on utilise gcc/g++. Enfin le compilateur passe au fichier contenant le main.

**L'édition de liens** Enfin, le compilateur agrège chaque fichier .o avec les éventuels fichiers binaires des librairies qui sont utilisées (fichiers .a et .so sous linux, fichiers .dll et .lib sous windows).

- Une librairie dynamique (.dll et .so) n'est pas recopiée dans l'exécutable final (ce qui signifie que le programme est plus petit et bénéficiera des mises à jour de ladite librairie). En contrepartie, la librairie doit être présente sur le système sur lequel tourne le programme.
- Une librairie statique (.a) est recopiée dans l'exécutable final ce qui fait que celui-ci est complètement indépendant des librairies installées du le système sur lequel il sera recopié. En contrepartie, l'exécutable est plus gros, il ne bénéficie pas des mises à jour de cette librairie etc...

Le linker vérifie en particulier que chaque fonction appelée dans le programme n'est pas seulement déclarée (ceci est fait lors de la compilation) mais aussi implémentée (chose qu'il n'avait pas vérifié à ce stade). Il vérifie aussi qu'une fonction n'est pas implémentée dans plusieurs fichiers .o.

Cette phase, appelée aussi édition de lien, constitue la phase finale pour obtenir un exécutable (noté .exe sous windows, en général pas d'extension sous linux).

Cette étape est assurée par ld lorsqu'on utilise gcc/g++.

## 1.2 Manipulation d'une librairie

### 1.2.1 Librairies statiques et dynamiques

Les librairies dynamiques permettent de ne pas incorporer le code de leurs éléments à l'exécutable lors de l'édition de liens. Le code de l'exécutable est donc plus petit. Au chargement de l'exécutable, le lien est établi avec les éléments requis dans les librairies partagées. Le code de ces librairies n'a qu'une seule copie en mémoire même s'il est référencé par plusieurs programmes à la fois.

Par défaut on utilise les librairies dynamiques.

Si on veut linker statiquement il faut forcer par l'option `-static` de *gcc*.

### 1.2.2 Différence d'utilisation des librairies statiques et dynamiques

Dans ce qui précède, nous n'avons fait aucune distinction entre les librairies statiques et dynamiques. En effet l'utilisation des deux types de librairies est presque identique. Il y a pourtant une petite différence : dans le cas des librairies dynamiques, si le programme allait toujours chercher les librairies au même emplacement, il suffirait de changer cet emplacement pour que le programme devienne inutilisable, ou qu'il faille le recompiler. C'est pourquoi pour chercher l'emplacement des librairies dynamiques, on s'aide d'une variable d'environnement appelée `LD_LIBRARY_PATH`.

Cette variable indique au programme à quels emplacements il doit chercher les librairies dynamiques. Si cet emplacement est modifié, il suffit de modifier la variable, sans changer le programme. Pour indiquer au système qu'il faut chercher dans le répertoire `/usr/local/lib`, il faudra initialiser la variable `LD_LIBRARY_PATH` de la manière suivante :

```
export LD_LIBRARY_PATH=/usr/local/lib
```

Si l'on veut que les programmes cherchent dans `/usr/local/lib`, dans `/usr/X11R6/lib` et dans le répertoire courant, il faudra écrire :

```
export LD_LIBRARY_PATH=.:/usr/X11R6/lib:/usr/local/lib
```

En pratique, vous ne définirez jamais cette variable mais vous ajouterez des fichiers à sa définition :

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/users/profs/habibi/libs
```

## 1.3 Compilation d'un code utilisant une librairie

### 1.3.1 Librairie statique

L'utilisation d'une bibliothèque statique est relativement triviale. Il suffit d'indiquer dans la ligne de commande de compilation le nom de la librairie à inclure dans l'option *-l* comme le montre l'exemple suivant pour une librairie nommée *libXXX.a* :

```
gcc ... -lXXX
```

### 1.3.2 Librairie dynamique

Il existe 2 manières d'utiliser une librairie dynamique : de façon implicite ou explicite. Le chargement implicite sera réalisé automatiquement par le système d'exploitation au démarrage de l'application. La compilation s'effectuant de la même manière que pour l'utilisation d'une librairie statique comme le montre l'exemple suivant pour une librairie nommée *libXXX.so* :

```
gcc ... -lXXX
```

Le chargement explicite est un petit peu plus fastidieux. Il nécessite de charger le fichier de la librairie et d'accéder aux fonctions contenues dans la librairie par l'intermédiaire de pointeurs de

fonction à initialiser aux bonnes adresses. L'ensemble de se travail est réalisé par l'intermédiaire de fonctions de la librairie `/lib/ld.so` (à compiler avec l'option de commande *-ldl*.

L'ensemble des fonctions de la librairie *ld* sont déclarées dans le fichier *dlfcn.h*. La fonction *dlopen()* permet de charger le fichier de librairie tandis que la fonction *dlsym()* permet de récupérer le pointeur de fonction associé à une fonction contenu dans la librairie. Une fois que les fonctions de la librairie ne sont plus utilisés, il est possible de libérer les ressources allouées en utilisant la fonction *dlclose()*.

L'exemple suivant montre comment charger depuis un exécutable de manière implicite une librairie nommée *malib.so* afin d'utiliser la fonction `void Initialize(unsigned int a,unsigned int b)` de cette librairie et finalement libérer les ressources utilisées.

```
void *lib;
typedef void (*pfInitialize)(unsigned int,unsigned int);
pfInitialize Initialize;

if ((lib=dlopen("malib.so",RTLD_LAZY))==NULL)
{
  // Erreur de chargement de la librairie
  return(false);
}
if ((Initialize=(pfInitialize)dlsym(lib,"Initialize"))==NULL)
  return(false);

// Utilisation de la fonction
Initialize(1,2);

// libération des ressources
dlclose(lib);
```

Attention, les librairies dynamiques chargées par *ld.so* sont cherchées dans l'ordre suivant :

— dans les répertoires indiqués dans la variable d'environnement `LD_LIBRARY_PATH` où ils sont séparés par ':'
— puis dans le fichier cache `/etc/ld.so.cache`
— puis dans les répertoires `/usr` et `/usr/lib`

Les librairies indiquées dans la variable d'environnement `LD_PRELOAD` sont chargées avant les autres et ont donc priorité. Leurs noms sont séparés par des blancs :

```
export LD_PRELOAD=$LD_PRELOAD /usr/local/lib/libstdc++-libc6.2-2.so.3
```

# 2 Création d'une librairie

## 2.1 Librairie statique

On le fait par la commande ar comme archive. Le nom de l'archive doit commencer par lib et se terminer par .a. On ajoute l'élément sub1.o dans l'archive libamoi.a par :

```
ar crv libamoi.a sub1.o
```

l'option `r` indique une insertion ou un remplacement, c indique la création de la librairie.

Effacement d'un élément :

```
ar d libamoi.a sub1.o
```

retire sub1.o de l'archive.

Extraction d'un élément :

```
ar x libamoi.a sub1.o
```

extrait sub1.o de l'archive.

Liste des éléments de l'archive :

```
ar t libamoi.a
```

Création d'un index des éléments

```
ranlib libamoi.a
```

Ceci accélère le link-edit.

Link-edit avec référence à une librairie statique personnelle :

```
cc -o prog prog.o -L. -lamoi
```

`-L` indique le répertoire où se trouve `libamoi.a`, `-lamoi` indique au linker d'aller chercher les références externes non satisfaites dans `libamoi.a` Créer une librairie dynamique personnelle.

### 2.2 Librairie dynamique

Pour créer une librairie dynamique sous Linux, il suffit d'utiliser l'option `shared` de gcc à une compilation traditionnelle d'exécutable. Il est également recommandé d'utiliser l'option `-fPIC` qui génère du code dont la position est indépendante d'une adresse fixée à la compilation.

```
gcc -fPIC -c *.c
gcc -shared -o libfoo.so.1 *.o
```

## 3 Exercice

À partir du TP précédant de gestion de liste, nous allons construire et utiliser les deux types de librairies : statiques et dynamiques.

Afin de ne pas perturber ni la compilation, ni l'exécution de vos programmes et librairies, utilisez systématiquement des répertoires différents pour la création puis l'utilisation des librairies.

Question 1. Reprenez la correction donnée pour le TP3 et compilez deux librairies statiques, avec et sans gestion d'allocation par bloc.

Question 2. Pour chaque version de librairie statique, créez un programme de test, par exemple à l'aide du fichier `main.c` fourni.

Question 3. Créez maintenant deux librairies dynamiques, avec et sans gestion d'allocation par bloc.

Question 4. Pour chaque version de librairie dynamique, créez un programme de test qui charge la librairie de manière implicite, par exemple à l'aide du fichier `main.c` fourni.

Question 5. Créez un nouveau programme de test qui charge à la demande explicitement une librairie dynamique de liste. Testez votre code pour chaque librairie dynamique créée.

Reprendre le dernier TP et créer une bibliothèque partagée contenant l'API de gestion de liste. L'utiliser ensuite dans un nouvel exécutable.