

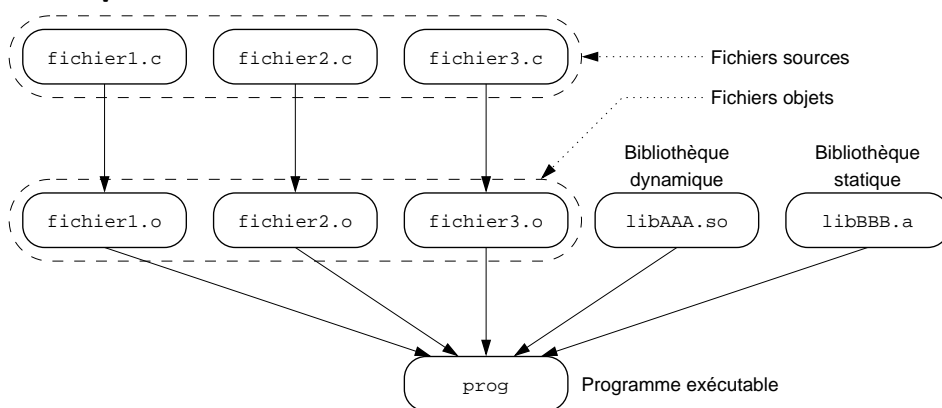
Mise en œuvre de bibliothèques

Compilation séparée
Recherche de symboles
Bibliothèques statiques
Bibliothèques dynamiques
Chargement dynamique

Fabrice HARROUET
 École Nationale d'Ingénieurs de Brest
 harrouet@enib.fr
 http://www.enib.fr/~harrouet/

Rappels sur la compilation séparée

▷ Principe



- ◇ Génération des fichiers objets (compilation) :
 - `$ cc -c -o fichier.o -Iinc_dir fichier.c`
- ◇ Génération du programme exécutable (édition de liens) :
 - `$ cc -o prog *.o -Llib_dir -lAAA -lBBB`

Rappels sur la compilation séparée

▷ Visibilité des symboles

- ◇ Les modules (`.o` , `.so` , `.a`) peuvent être interdépendants
 - Des *symboles* sont importés/exportés dans chacun d'eux
- ◇ Bibliothèque \equiv ensemble de fichiers objets (`.o`)
 - Symboles d'une bibliothèque \equiv ensemble des symboles des `.o`
- ◇ Édition de liens
 - Association des symboles importés et exportés
 - Vérification des doublons
 - Placement en mémoire
 - Données et traitements de démarrage
 - ...

Rappels sur la compilation séparée

▷ Visibilité des symboles

- ◇ Symboles accessibles : les fonctions, les variables globales
 - Par défaut tout est exporté (sauf sous *Window\$*)
 - Définition `static` \rightarrow portée locale au `.o` courant
 - Importation d'un symbole par son accès
(`@/`appel d'une fonction, `@/`lecture/écriture d'une variable)
- ◇ Déclaration \neq définition
 - `extern int i;` (déclaration d'une variable globale)
 - `int i;` (définition d'une variable globale)
 - `void f(int i);` (déclaration d'une fonction)
 - `void f(int i) { ... }` (définition d'une fonction)

Rappels sur la compilation séparée

- ▷ **Visibilité des symboles en langage C**
 - ◇ Nom d'un symbole \equiv nom de la variable ou de la fonction
 - ◇ Aucune ambiguïté, information suffisante
- ▷ **Visibilité des symboles en langage C++**
 - ◇ Variables globales : pas de problème
 - ◇ Fonctions : surcharge possible (plusieurs signatures pour le même nom)
 - Codage (non standard) de la signature dans le nom de symbole
 - ◇ Déclaration **extern "C"** d'une fonction
 - Nom de symbole \equiv nom de la fonction
 - Autorisé une seule fois pour un symbole donné
 - Nécessaire pour utiliser une bibliothèque écrite en C dans un programme écrit en C++
(`#ifndef __cplusplus`)

Recherche de symboles dans les modules

- ▷ **L'utilitaire nm** (man 1 nm)
 - ◇ Disponible dans tout environnement de développement
 - Compilateur, assembleur, *linker*, débogueur ...
 - ◇ Affiche des informations concernant les symboles
 - Des fichiers objets
 - Des bibliothèques statiques
 - Des bibliothèques dynamiques
 - Des fichiers exécutables
 - ◇ Utilisation classique
 - "Il manque un symbole à l'édition de liens !"
 - "Un symbole est défini à multiples reprises !"
 - On le recherche parmi dans les bibliothèques disponibles

Recherche de symboles dans les modules

▷ L'utilitaire nm

- ◇ `$ nm [options] module`
- ◇ Options dépendantes du système
 - `-B` : affichage style *BSD* (standard)
 - `-D` : pour bibliothèques dynamiques
 - `-C` : décodage des symboles *C++*
- ◇ Sortie : trois colonnes
 - Adresse, type (`T t D d B b U ...`) et nom du symbole
- ◇ Exemple d'utilisation :


```
$ for i in lib* ; do
>   if ( nm -BD $i | grep "MON_SYMBOLE" ) ; then
>     echo "---> $i"
>   fi
> done
```

Recherche de symboles dans les modules

```
int SYMB_VAR1=100;
int SYMB_VAR2;
extern int SYMB_VAR3;
static int SYMB_VAR4=200;
static int SYMB_VAR5;

int SYMB_FNCT1(int i)
{ return i+SYMB_VAR1; }

extern "C" int SYMB_FNCT2(int i)
{ return i+SYMB_VAR2; }

extern int SYMB_FNCT3(int i);

static int SYMB_FNCT4(int i)
{
SYMB_VAR2=SYMB_FNCT1(1234);
SYMB_VAR3=SYMB_FNCT2(1234);
SYMB_VAR4=SYMB_FNCT3(1234);
SYMB_VAR5=SYMB_FNCT4(1234);
return i;
}
```

```
$ nm prog.o | grep SYMB
00000000 T SYMB_FNCT1__Fi
0000001c T SYMB_FNCT2
                U SYMB_FNCT3__Fi
00000038 t SYMB_FNCT4__Fi
00000000 D SYMB_VAR1
00000000 B SYMB_VAR2
                U SYMB_VAR3
00000004 d SYMB_VAR4
00000004 b SYMB_VAR5
```

```
T: TEXT global    t: TEXT local
D: DATA global   d: DATA local
B: BSS global     b: BSS local
U: undefined
```

Les bibliothèques statiques

▷ Utilisation d'une bibliothèque statique

- ◇ Généralement de la forme `libXXX.a`
- ◇ Une simple collection de fichiers objets sur le même thème
 - Inspectée à l'édition de liens
 - Modules objets requis incorporés à l'exécutable
 - Uniquement les modules nécessaires (découper au maximum)
- ◇ Édition de liens :
 - `$ cc ... -lXXX` (pour `libXXX.a`)
 - Problème : recherche des bibliothèques dynamiques avant
 - Utiliser l'option `-static` → pas de bibliothèques dynamiques
 - Indiquer la bibliothèque "en dur" : `$ cc/libXXX.a`
(considérée comme la liste de ses `.o`)

Les bibliothèques statiques

▷ Avantages d'une bibliothèque statique

- ◇ L'exécutable est complètement autonome
- ◇ Édition de lien réussie → fonctionnement immuable

▷ Inconvénients d'une bibliothèque statique

- ◇ Duplication des modules dans l'exécutable
 - Consommation d'espace disque (gros exécutables)
 - Consommation de mémoire (pas de partage inter-processus)
- ◇ Exécutable insensible aux mises à jour des bibliothèques utilitaires

Les bibliothèques statiques

▷ Réalisation d'une bibliothèque statique

- ◇ Utiliser la commande `ar` (`man 1 ar`)
- ◇ De nombreuses options ...
- ◇ Commande usuelle : `$ ar cr libXXX.a *.o`
 - `c` : créer à partir de rien
 - `r` : insertion inconditionnelle
- ◇ Possibilités d'ajout, de retrait, de mise à jour ...
- ◇ Création d'un index : `$ ranlib libXXX.a`
 - Accélère l'édition de liens
 - Généralement facultatif (sauf sous *MacOSX*)

Les bibliothèques dynamiques

▷ Utilisation d'une bibliothèque dynamique

- ◇ Généralement de la forme `libXXX.so`
- ◇ Une simple collection de fichiers objets sur le même thème
 - Inspectée à l'édition de liens
 - Références aux symboles incorporés à l'exécutable
- ◇ Édition de liens :
 - `$ cc ... -lXXX` (pour `libXXX.so`)
 - Bibliothèques dynamiques recherchées avant les statiques
 - Indiquer la bibliothèque "en dur" : `$ cc/libXXX.so`

Les bibliothèques dynamiques

▷ Avantages d'une bibliothèque dynamiques

- ◇ Pas de duplication du code
 - Fichiers exécutables de petite taille
 - Chargées une seule fois en mémoire (partagées entre tous les processus qui en ont besoin)
- ◇ Exécutables sensibles aux mises à jour

▷ Inconvénients d'une bibliothèque dynamiques

- ◇ Être capable de retrouver la bibliothèque au lancement
- ◇ Possibilité de “*casser*” d'anciens programmes

Les bibliothèques dynamiques

▷ Réalisation d'une bibliothèque dynamique

- ◇ Utiliser le compilateur habituel avec l'option `-shared`
 - Exactement comme la génération d'un exécutable
 - Génère uniquement une bibliothèque dynamique
 - Pas besoin de `main()` ...
- ◇ Exemple :
`$ cc -shared -o libXXX.so *.o`
- ◇ Réduire la taille de la bibliothèque (`man 1 strip`)
 - Retirer les symboles inutiles (débogages ...)
 - Commande : `$ strip -s libXXX.so`

Les bibliothèques dynamiques

▷ Retrouver les bibliothèques dynamiques au lancement

- ◇ Préciser des répertoires non-standards
 - \$ export LD_LIBRARY_PATH=\$HOME/lib:/usr/local/XXX/lib
 - Utile pendant la phase de développement de la bibliothèque
 - Solution temporaire, installation finale dans un répertoire standard
- ◇ Répertoires standards implicites (/usr/lib, /usr/X11R6/lib ...)

▷ Le préchargement de symboles

- ◇ Forcer le préchargement des symboles de bibliothèques spécifiques
- ◇ \$ LD_PRELOAD="./libXXX.so ./libYYY.so" ./prog
- ◇ prog ne cherche pas ces symboles dans les autres bibliothèques
- ◇ Permet de choisir une implémentation particulière parmi plusieurs
 - Mesures de performances, débogage ...

Les bibliothèques dynamiques

▷ Lister les dépendances

- ◇ Commande ldd
- ◇ Affiche les bibliothèques dynamiques associées au chargement
 - d'un programme dynamiquement lié
 - d'une bibliothèque dynamique

```
$ ldd ./prog
libfunction.so => not found
libstdc++-libc6.1-2.so.3 => /usr/lib/libstdc++-libc6.1-2.so.3 (0x40019000)
libm.so.6 => /lib/libm.so.6 (0x40068000)
libc.so.6 => /lib/libc.so.6 (0x40086000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)

$ export LD_LIBRARY_PATH=.
$ ldd ./prog
libfunction.so => ./libfunction.so (0x40014000)
libstdc++-libc6.1-2.so.3 => /usr/lib/libstdc++-libc6.1-2.so.3 (0x4001b000)
libm.so.6 => /lib/libm.so.6 (0x4006a000)
libc.so.6 => /lib/libc.so.6 (0x40088000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

Mise en œuvre de bibliothèques dynamiques

```

$ g++ -c f1.cpp
$ g++ -c f2.cpp
$ g++ -c f3.cpp
$ g++ -shared -o libfunction.so f1.o f2.o f3.o
$ g++ -c prog.cpp
$ g++ -o prog prog.o -L. -lfunction
$ ./prog
./prog: error in loading shared libraries: libfunction.so: cannot open
shared object file: No such file or directory
$ export LD_LIBRARY_PATH=.
$ ./prog
f1()
f2()
f3()
$ g++ -c newf1.cpp
$ g++ -c newf3.cpp
$ g++ -shared -o libnewf1f3.so newf1.o newf3.o
$ LD_PRELOAD=./libnewf1f3.so ./prog
new f1()
f2()
new f3()
$

```

Chargement dynamique de symboles

▷ Principe

- ◇ Charger des ressources (code et données) à la demande
 - Ressources initialement inconnues
 - Choisies lors de l'exécution du programme
 - Dans des bibliothèques dynamiques
 - Chargées et déchargées explicitement
- ◇ ≠ édition de liens habituelle
 - Liste de bibliothèques à charger au démarrage
 - Chargées implicitement
 - Chargées pour toute la durée de l'exécution
- ◇ Donner des possibilités d'extension à une application
 - Principe des *plug-ins*

Chargement dynamique de symboles

▷ Démarche de mise en œuvre

- ◇ `#include <dlfcn.h>` (man 3 `dlopen`)
- ◇ Ouverture avec `dlopen()` (`dlerror()` en cas d'échec)
- ◇ Accès aux symboles avec `dlsym()` (`dlerror()` en cas d'échec)
- ◇ Fermeture avec `dlclose()`
- ◇ Les symboles obtenus avec `dlsym()` deviennent inaccessibles

▷ Réalisation du programme initial

- ◇ Édition de liens avec `-ldl`
- ◇ Édition de liens avec l'option `-rdynamic` (sous *Linux*)
 - Permet au code du *plugin* d'accéder au code initial
 - C'est généralement nécessaire
 - Sans cette option il faudrait mettre le code initial dans une bibliothèque dynamique pour la lier au *plugin*

Chargement dynamique de symboles

▷ La fonction `dlopen()`

- ◇ `void * dlopen(const char * path,int mode);`
- ◇ `path` : bibliothèque à charger dans l'espace d'adressage
 - Recherche classique ou chemin absolu/relatif
 - Pointeur nul → code initial (global)
- ◇ `mode` : mode d'ouverture (ou bit à bit)
 - `RTLD_LAZY` : ne résoudre qu'à l'appel des fonctions
 - `RTLD_NOW` : résoudre toutes les fonctions au chargement
 - `RTLD_GLOBAL` : symboles insérés dans le code initial
 - `RTLD_LAZY` et `RTLD_NOW` sont exclusifs
 - Démarche prudente généralement adoptée : `RTLD_LAZY`
- ◇ Retour : pointeur sur la bibliothèque chargée (nul si erreur)
- ◇ Erreurs : voir `dlerror()` plus loin

Chargement dynamique de symboles

▷ La fonction `dlsym()`

- ◇ `void * dlsym(void * handle, const char * name);`
- ◇ Recherche le symbole **name** dans la bibliothèque **handle**
- ◇ **handle** : préalablement obtenu par `dlopen()`
- ◇ **name** : nom du symbole tel qu'affiché par **nm**
 - Penser à **extern "C"** pour les fonctions en *C++*
- ◇ Retour : pointeur sur le symbole (nul si erreur)
- ◇ Erreurs : voir `dlerror()` plus loin
- ◇ Aucune vérification de type !
 - Convertir en pointeur sur le type de donnée attendue
 - Une donnée ? De quel type ?
 - Une fonction/méthode ? Quelle signature ?
- ◇ nb : pour *ISO C* @fonction \neq `void * !!!` → `memcpy()`

Chargement dynamique de symboles

▷ La fonction `dlerror()`

- ◇ `char * dlerror(void);`
- ◇ Fournit un message d'erreur sur la dernière opération `dl ??? ()`
- ◇ Retour : le message **alloué statiquement** ou pointeur nul si ok
- ◇ Recopier ou afficher la chaîne immédiatement
- ◇ N'invoquer qu'une fois de suite (deuxième fois → ok)

▷ La fonction `dlclose()`

- ◇ `int dlclose(void * handle);`
- ◇ Déréférence **handle** préalablement obtenu par `dlopen()`
- ◇ Cette bibliothèque n'est plus accessible par le processus
- ◇ Retour : **0** si ok, sinon voir `dlerror()`

Chargement dynamique de symboles

```

$ cat prog.cpp
#include <dlfcn.h>
#include <stdio.h>
#include <string.h>
int nb=0;
typedef void (*FnctPtr)(void);
int main(void)
{
void * lib=dlopen("libdltest.so",RTLD_LAZY);
if(lib)
{
void * symb=dlsym(lib,"value");
if(symb) *((int *)symb)=0xDEADBEEF;
else fprintf(stderr,"%s\n",dlerror());
symb=dlsym(lib,"function");
FnctPtr fnct;
memcpy(&fnct,&symb,sizeof(void*));
if(fnct) (*fnct)();
else fprintf(stderr,"%s\n",dlerror());
dlclose(lib);
}
else fprintf(stderr,"%s\n",dlerror());
return 0;
}
$ cat dltest.cpp
#include <stdio.h>
extern int nb; // dans prog.cpp
int value=0;
extern "C" void function(void)
{
printf("function() value=%x\n",value);
nb++;
}
$ g++ -o prog prog.cpp -ldl -rdynamic
$ g++ -shared -o libdltest.so dltest.cpp
$ ./prog
libdltest.so: cannot open shared object
file: No such file or directory
$ export LD_LIBRARY_PATH=.
$ ./prog
function() value=deadbeef
$

```

Chargement dynamique de symboles

```

#include <dlfcn.h>
#include <stdio.h>

int nb=0;

int main(int argc,char ** argv)
{
void * global=dlopen((const char *)0,RTLD_LAZY);
void * lib=dlopen("libdltest.so",RTLD_LAZY | (argc>1 ? RTLD_GLOBAL : 0));
if(lib)
{
void * symb=dlsym(lib,"function");
if(symb) fprintf(stderr,"symbol found in library\n");
else fprintf(stderr,"%s\n",dlerror());
symb=dlsym(global,"function");
if(symb) fprintf(stderr,"symbol found in global context\n");
else fprintf(stderr,"%s\n",dlerror());
dlclose(lib);
}
else fprintf(stderr,"Loading %s\n",dlerror());
dlclose(global);
return 0;
}

```

Chargement dynamique de symboles

▷ Effet de `-rdynamic` (*Linux*) et `RTLD_GLOBAL`

```
$ export LD_LIBRARY_PATH=.
$
$ g++ -o prog prog.cpp -ldl
$
$ ./prog
Loading ./libdltest.so: undefined symbol: nb
$
$ g++ -o prog -rdynamic prog.cpp -ldl
$
$ ./prog
symbol found in library
./prog: undefined symbol: function
$
$ ./prog RTLD_GLOBAL
symbol found in library
symbol found in global context
$
```

Spécificités de certains systèmes

▷ Sous *MacOSX*

- ◇ `ranlib` obligatoire pour les bibliothèques statiques
- ◇ Nom standard des bibliothèques dynamiques : `libXXX.dylib`
- ◇ Bibliothèque dynamique \neq *plugin* (*bundle*)


```
$ g++ -flat_namespace -o prog ...
$ g++ -flat_namespace -dynamiclib -o libXXX.dylib ...
$ g++ -flat_namespace -bundle -o libXXX.dylib ...
```
- ◇ Pas d'option `-rdynamic` → mettre l'appli dans une bibliothèque
- ◇ `otool -L binaire` au lieu de `ldd binaire`
- ◇ `DYLD_LIBRARY_PATH` au lieu de `LD_LIBRARY_PATH`
(`DYLD_LIBRARY_PATH` → répertoires standards → .)
- ◇ `DYLD_INSERT_LIBRARIES` au lieu de `LD_PRELOAD`

Spécificités de certains systèmes

▷ Sous *Windows*

- ◇ Nom standard des bibliothèques statiques : *XXX.lib*
> `lib.exe /out:XXX.lib XXX.obj`
- ◇ Nom standard des bibliothèques dynamiques : *XXX.dll*
- ◇ Édition de liens avec un fichier *.lib* (pas *.dll* !) créé en même temps
> `link.exe /dll /out:XXX.dll /implib:XXX.lib XXX.obj`
> `link.exe /out:prog.exe prog.obj XXX.lib`
- ◇ Pas d'option `-rdynamic` → mettre l'appli dans une bibliothèque
- ◇ `dumpbin.exe /exports binaire` au lieu de `nm binaire`
- ◇ `dumpbin.exe /dependents binaire` au lieu de `ldd binaire`
(chemins non indiqués !)
- ◇ `PATH` au lieu de `LD_LIBRARY_PATH`
(répertoire du *.exe* → répertoires `C:\WINDOWS\...` → `.` → `PATH`)
- ◇ Pas de `LD_PRELOAD` (éventuellement *Detours* mais modif. binaires !)

Spécificités de certains systèmes

▷ Sous *Windows*

- ◇ Pas d'export/import par défaut dans les bibliothèques dynamiques !
- ◇ Exportation explicite pour réaliser la bibliothèque :
 - `__declspec(dllexport) void MyFunct(void);`
 - `class __declspec(dllexport) MyClass { ... };`
- ◇ Importation explicite pour utiliser la bibliothèque :
 - `__declspec(dllimport) void MyFunct(void);`
 - `class __declspec(dllimport) MyClass { ... };`
- ◇ Nécessiterait de faire les *.h* en double !!!
 - Utilisation d'un jeu de *macros* astucieux
 - Contrôlé depuis le `makefile` (ou autre)

Spécificités de certains systèmes

▷ Sous Window\$

- ◇ Exemple de fichier `.h` pour une bibliothèque dynamique (ou un *plugin*)

```
#ifndef _WIN32
#  ifdef WIN32_EXAMPLE_DLL
#    define WIN32_EXAMPLE_API __declspec(dllexport)
#  else
#    define WIN32_EXAMPLE_API __declspec(dllimport)
#  endif
#else
#  define WIN32_EXAMPLE_API
#endif

class WIN32_EXAMPLE_API MyClass // classe exportee par la bibliotheque
{ /* ... */ };

WIN32_EXAMPLE_API void myFunction(void); // fonction exportee par la bibliotheque

extern "C" WIN32_EXAMPLE_API void myPluginFnct(void); // point d'entree d'un plugin
```

enib, F.H ... 29/31

Spécificités de certains systèmes

▷ Sous Window\$

- ◇ Réalisation/utilisation d'une bibliothèque dynamique
- ◇ Les fichiers `.cpp` incluent le `.h` précédent
- ◇ Option `/D` utilisée ou non à la compilation
 - influencer notre *macro* `WIN32_EXAMPLE_API`

```
# realisation de la bibliotheque dynamique example.dll
> cl.exe /c /DWIN32_EXAMPLE_DLL /FoexA.obj exA.cpp
> cl.exe /c /DWIN32_EXAMPLE_DLL /FoexB.obj exB.cpp
> link.exe /dll /out:example.dll /implib:example.lib exA.obj exB.obj

# realisation du programme prog utilisant de la bibliotheque dynamique example.dll
> cl.exe /c /Foprog.obj prog.cpp
> link.exe /out:prog.exe prog.obj example.lib
```

enib, F.H ... 30/31

Spécificités de certains systèmes

▷ Sous *Windows*

- ◇ LoadLibrary(), GetProcAddress(), FreeLibrary()
au lieu de dlopen(), dlsym(), dlclose()

```
#include <stdio.h>
#include <windows.h>
int main(void)
{
    HINSTANCE lib=LoadLibrary("example.dll");
    if(lib)
    {
        void (*fct)(void)=(void (*)(void))GetProcAddress(lib,"myPluginFunction");
        if(fct) (*fct)();
        else fprintf(stderr,"GetProcAddress: %d\n",GetLastError());
        FreeLibrary(lib);
    }
    else fprintf(stderr,"LoadLibrary: %d\n",GetLastError());
    return 0;
}
```