I3 – Algorithmique et programmation
Introduction à la programmation en langage C
Cours n°7
Les bibliothèques, compilation avec Make

Camille Coti
camille.coti@iutv.univ-paris13.fr

IUT de Villetaneuse, département R&T

2011 – 2012

Intérêt d'une bibliothèque

Définition

Une bibliothèque contient **l'implémentation** de fonctions et procédures qui peuvent être appelées depuis d'autres programmes. C'est du **code objet** qui est lié au programme au moment de la phase d'**édition de liens** de la compilation.

Intérêts :

- Implémentation réalisée une fois pour toutes
- Réutilisation de fonctions et procédures génériques dans plusieurs programmes

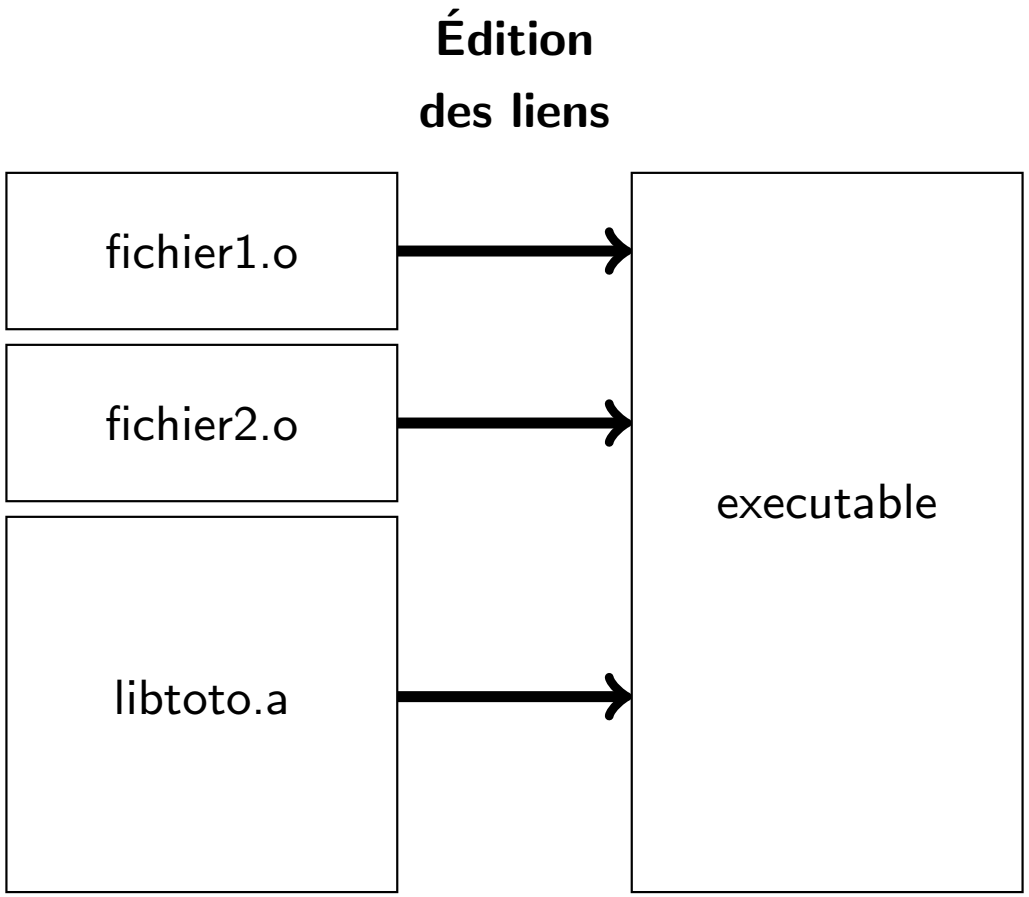
Exemple : la fonction `sqrt()` de la bibliothèque mathématique `libm`

- De nombreux programmes ont besoin de calculer une racine carrée
- On l'a implémentée une seule fois, dans une bibliothèque
- N'importe quel programme peut appeler cette fonction en utilisant cette bibliothèque

Bibliothèque statique

Définition

Une **bibliothèque statique** (extension **.a**) est une bibliothèque dont le code objet est directement **intégré à l'exécutable** lors de la compilation.



Bibliothèque statique

On peut examiner les symboles exportés par une bibliothèque statique avec la commande **nm** :

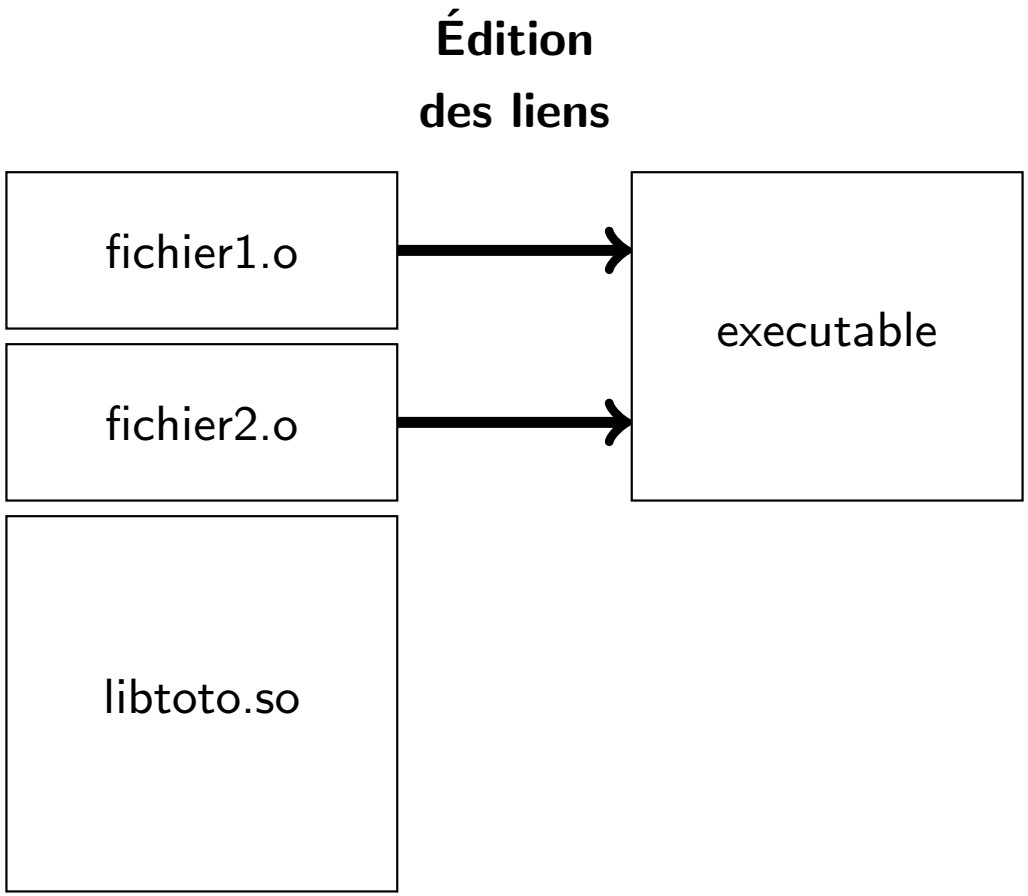
```
1  $ nm /users/coti/tools/lib/libplasma.a
2  allocate.o:
3           U free
4           U malloc
5           U plasma_element_size
6           U plasma_error
7  00000040 T plasma_private_alloc
8  00000000 T plasma_private_free
9  00000090 T plasma_shared_alloc
10 00000020 T plasma_shared_free
11 ...
```

- **T** : le symbole est défini dans la bibliothèque ; **U** : le symbole n'est pas défini dans la bibliothèque (la bibliothèque lui fait appel)
- 1ere colonne : adresse dans la bibliothèque où il est défini
- ligne 2 : nom du fichier objet où les symboles apparaissent (la bibliothèque étant ici faite de plusieurs fichiers objets rassemblés)

Bibliothèque dynamique

Définition

Une **bibliothèque dynamique** (extension **.so**) est une bibliothèque dont le code objet est **lié** à l'exécutable.



Bibliothèque dynamique

On peut également examiner les symboles exportés par une bibliothèque dynamique avec la commande **nm** :

```
$ nm /users/coti/tools/lib/libgoto2.so
0005dc10 T caxpy_
005aeac0 T cbdsqr_
0005e960 T cblas_caxpy
0005eb00 T cblas_ccopy
0005eb80 T cblas_cdotc
0005ec10 T cblas_cdotu
00063c90 T cblas_cgbmv
00068a50 T cblas_cgemm
00062ff0 T cblas_cgemv
00063510 T cblas_cgerc
```

Differences statique / dynamique

Taille de l'exécutable produit

- Un éxécutable utilisant une bibliothèque statique **contient le code de la bibliothèque** : il est par conséquent plus gros qu'un exécutable utilisant une bibliothèque dynamique.
- D'un point de vue global : la bibliothèque statique est **dupliquée** dans tous les exécutables qui l'utilisent ; une bibliothèque dynamique n'est présente qu'une fois sur le système.

Dépendances

- Un exécutable utilisant une bibliothèque statique contient tout ce dont il a besoin pour s'exécuter.
- Un exécutable utilisant une bibliothèque dynamique aura besoin que cette bibliothèque soit présente sur le système pour s'exécuter.

Maintenance

- Si un bug est corrigé dans une bibliothèque statique, les exécutables l'utilisant devront être recompilés.
- Si une bibliothèque dynamique est modifiée, les exécutables qui l'utilisent utiliseront automatiquement la version modifiée.

Compilation avec une bibliothèque

Exemple : utilisation de la fonction `sqrt()`, définie dans le fichier `math.h` et implémentée dans la bibliothèque `libm` :

- Inclusion du fichier d'en-têtes où la fonction est définie :

```
1 #include <math.h>
```

- Compilation avec -l<nom de la lib sans `lib`>
  - Ici : la lib s'appelle libm.so ou libm.a → -lm

```
1 $ gcc -o executable source1.c source2.c -lm
```

- Si la bibliothèque ne se trouve pas dans les répertoires standards : on précise le répertoire avec l'option -L<chemin> (sans espace)

```
1 $ gcc -o executable source1.c source2.c -L/usr/lib -lm
```

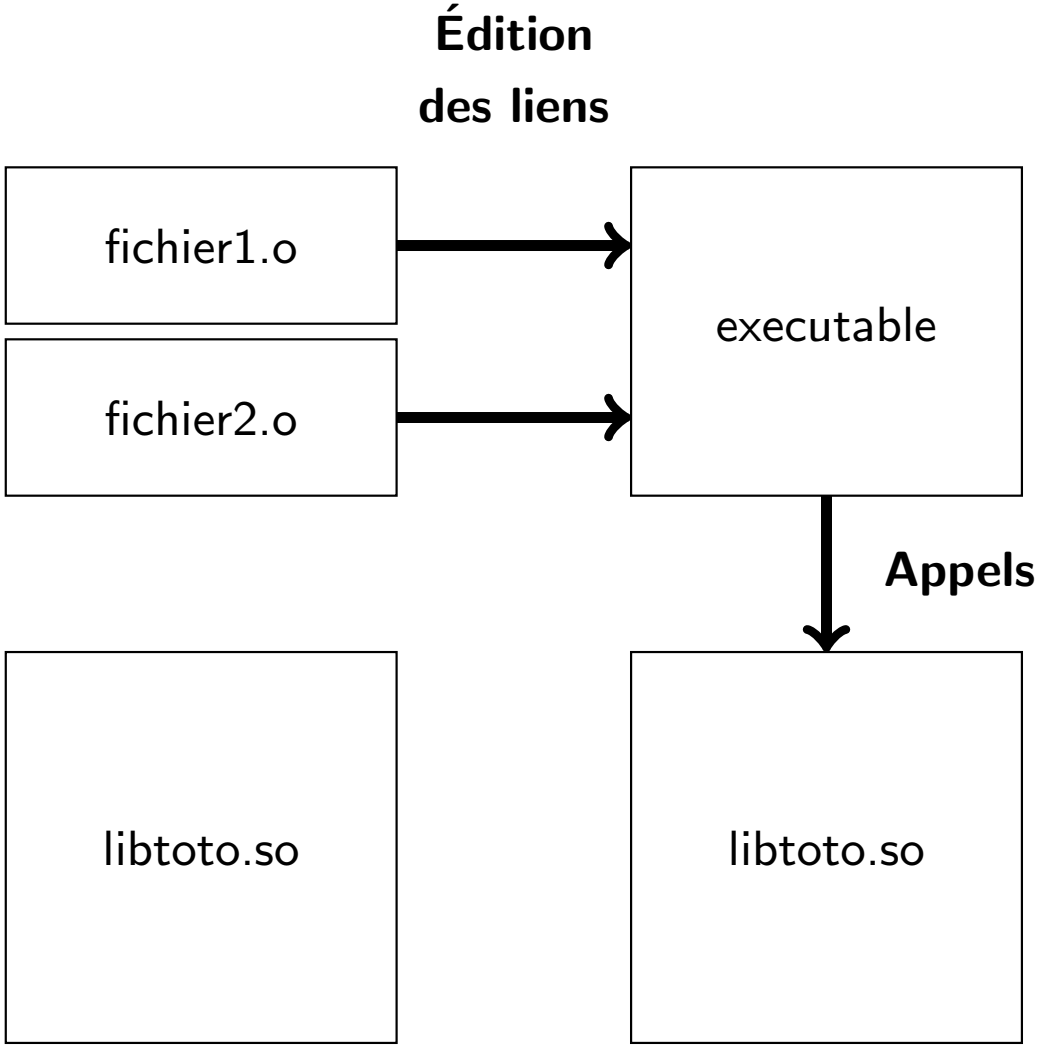
- Si le fichier d'en-têtes inclus ne se trouve pas dans les répertoires standards : on précise le répertoire avec l'option -I<chemin> (sans espace)

```
1 $ gcc -o executable source1.c source2.c -I/usr/include
```

Bibliothèque dynamique : Édition des liens à la compilation

Dernière étape de la compilation : édition des liens

- Liage entre eux des fichier objet obtenus à partir des fichiers sources
- Création de liens avec les bibliothèques dynamiques



Bibliothèque dynamique : Édition des liens à la compilation

Juste après la compilation, les références vers les définitions des fonctions ne sont **pas** résolues

- L'éditeur de liens vérifie juste que l'implémentation de la fonction est présente
- Si l'implémentation n'est pas trouvée : `undefined reference to ...` et échec de la compilation à l'étape de l'édition des liens

```
1  $ ldd xtest
2  linux-gate.so.1 =>  (0xb7786000)
3  libmpi_f77.so.1 => not found
4  libmpi.so.1 => not found
5  libdl.so.2 => /lib/i686/cmov/libdl.so.2 (0xb75f7000)
6  libnsl.so.1 => /lib/i686/cmov/libnsl.so.1 (0xb75e0000)
7  libutil.so.1 => /lib/i686/cmov/libutil.so.1 (0xb75dc000)
8  libgfortran.so.3 => /usr/lib/libgfortran.so.3 (0xb7517000)
9  libm.so.6 =>  /lib/i686/cmov/libm.so.6 (0xb74f0000)
10 libgcc_s.so.1 => /lib/libgcc_s.so.1 (0xb74d2000)
11 libpthread.so.0 => /lib/i686/cmov/libpthread.so.0 (0xb74b9000)
12 libc.so.6 => /lib/i686/cmov/libc.so.6 (0xb7373000)
13 /lib/ld-linux.so.2 (0xb7787000)
```

`libmpi_f77.so.1, libmpi.so.1` : pas trouvées pour l'instant

Bibliothèque dynamique : Résolution des liens à l'exécution

Les liens non établis à la compilation sont résolus lors de la première exécution

- On va chercher les bibliothèques dans les chemins indiqués dans la variable d'environnement `$LD_LIBRARY_PATH`
- Pour installer une bibliothèque dans un répertoire donné : ajout du chemin dans la variable `$LD_LIBRARY_PATH`

```
1  $ ldd xtest
2  linux-gate.so.1 =>  (0xb7786000)
3  libmpi_f77.so.1 => /users/coti/tools/lib/libmpi_f77.so.1 (0xb7757000)
4  libmpi.so.1 => /users/coti/tools/lib/libmpi.so.1 (0xb7623000)
5  libdl.so.2 => /lib/i686/cmov/libdl.so.2 (0xb75f7000)
6  libnsl.so.1 => /lib/i686/cmov/libnsl.so.1 (0xb75e0000)
7  libutil.so.1 => /lib/i686/cmov/libutil.so.1 (0xb75dc000)
8  libgfortran.so.3 => /usr/lib/libgfortran.so.3 (0xb7517000)
9  libm.so.6 => /lib/i686/cmov/libm.so.6 (0xb74f0000)
10 libgcc_s.so.1 => /lib/libgcc_s.so.1 (0xb74d2000)
11 libpthread.so.0 => /lib/i686/cmov/libpthread.so.0 (0xb74b9000)
12 libc.so.6 => /lib/i686/cmov/libc.so.6 (0xb7373000)
13 /lib/ld-linux.so.2 (0xb7787000)
```

- `libmpi_f77.so.1, libmpi.so.1` ont été trouvées dans `/users/coti/tools/lib/`
- `/users/coti/tools/lib/` est dans la variable `$LD_LIBRARY_PATH`

Interface

Les définitions nécessaires à l'utilisation de la bibliothèque sont décrites dans un fichier d'en-têtes (extension `.h`)

- Définitions (prototypes) des fonctions et des procédures implémentées par la bibliothèque ;
- Définitions des structures et des types de données manipulés ;
- Définition des constantes.

Définition : partie publique / partie privée

Ce fichier constitue la **partie publique** de la bibliothèque : l'utilisateur peut la lire pour comprendre comment l'utiliser.
À l'inverse, la bibliothèque elle-même (fichier .so ou .a) est la **partie privée** : c'est du code objet, l'utilisateur ne peut pas lire ce qu'elle contient.

Ce fichier d'en-tête constitue **l'interface** : c'est la liste des fonctions et des structures qui vont être appelées ou utilisées par les programmes utilisant la bibliothèque.

Création d'une bibliothèque statique

- Génération de code objet de tous les fichiers qui vont être utilisés dans la bibliothèque

```
1 $ gcc -c fichier1.c
2 $ gcc -c fichier2.c
```

On obtient des fichiers `.o` contenant le code objet

- Création de la bibliothèque en assemblant tous les fichiers de code objet dans une seule archive avec la commance `ar` et l'option `r` :

```
1 $ ar r libmabiblio.a fichier1.o fichier2.o
```

On obtient un fichier `libmabiblio.a`.

- On peut lister les fichiers objets inclus dans l'archive avec l'option `t` de la commande `ar`

```
1 $ ar libmabiblio.a
2 fichier1.o
3 fichier2.o
```

- Enfin, on crée l'index des symboles définis dans la bibliothèque (fonctions implémentées) et on le stocke dans l'archive avec la commande `ranlib` :

```
1 $ ranlib libmabiblio.a
```

Création d'une bibliothèque dynamique

- Génération de code objet de tous les fichiers qui vont être utilisés dans la bibliothèque (comme pour une bibliothèque statique)

```
1 $ gcc -c fichier1.c
2 $ gcc -c fichier2.c
```

On obtient des fichiers `.o` contenant le code objet

- Création de la bibliothèque avec `gcc` :
  - Option `-shared` : bibliothèque partagée
  - Option `-o libmalib.so` : nom de la bibliothèque

```
1 $ gcc -shared -o libmalib.so fichier1.o fichier2.o
```

On obtient le fichier `libmalib.so`

Présentation de GNU Make

GNU Make est un utilitaire de compilation

- Il exécute des **suites de commandes** définies dans un fichier
- On peut **automatiser la compilation**
- Il **gère des dépendances** entre des fichiers
- Il permet de **ne recompiler que ce qui est nécessaire**

Exemple : j'ai un exécutable `exec` qui est compilé à partir des fichiers sources `src1.c` et `src2.c`

- On compile `src1.c` et `src2.c` pour obtenir les fichiers objets `src1.o` et `src2.o`
- Puis on effectue l'édition des liens pour obtenir `exec`
- On modifie `src1.c`
- Pour recompiler `exec` on n'a besoin que de recompiler `src1.o` : `src2.o` n'a pas été modifié, pas besoin de el recompiler

Utilisation de Make

- La commande s'appelle `make`.

```
1 $ make
```

- Le programme `make` lit un fichier où sont définies les commandes à exécuter : le **makefile**
  - Par défaut on lit le fichier `makefile` ou `Makefile` situé dans le répertoire courant ;
  - Si on veut utiliser un autre fichier : option `-f`

```
1 $ make -f ../fichier
```

Autre option utile : `make -n` : afficher la suite d'actions à effectuer mais ne rien faire

```
1 $ make -n
```

Écriture d'un Makefile

Le Makefile est organisé en **sections**, composées :

- d'une cible ;
- de dépendances ;
- d'une ou plusieurs actions à effectuer.

La syntaxe est la suivante :

```
1 cible : dependances
2         action
```

Attention : les lignes d'action doivent commencer par une tabulation.
Exemple :

```
1 exec : source1.o source2.o
2         gcc -o exec source1.o source2.o
```

- La cible `exec` dépend des fichiers objets `source1.o` et `source2.o`.
- Si ces deux fichiers sont présents et si `exec` n'existe pas ou qu'il est plus ancien que `source1.o` et `source2.o` : on compile `exec` en exécutant l'action à effectuer.

Dépendances entre les cibles

Lorsqu'une dépendance n'est pas présente

- On cherche une cible dont le nom correspond à la dépendance
- On l'exécute
- Et ainsi de suite, récursivement, jusqu'à ce que toutes les dépendances soient présentes

Attention aux noms des cibles !

- Ils doivent correspondre aux noms des dépendances

Dépendances entre les cibles

Exemple :

```
1 exec : source1.o source2.o
2 	gcc -o exec source1.o source2.o
3
4 source1.o: source1.c entetes.h
5 	gcc -c source1.c
6
7 source2.o: source2.c entetes.h
8 	gcc -c source2.c
```

On veut compiler `exec` et `source1.o` et `source2.o` ne sont pas présents :

- On va voir la cible `source1.o` : elle dépend de `source1.c` et `entetes.h`
- On exécute l'action : on compile `source1.c` pour obtenir le fichier de code objet `source1.o`
- On va voir la cible `source2.o` : elle dépend de `source2.c` et `entetes.h`
- On exécute l'action : on compile `source2.c` pour obtenir le fichier de code objet `source2.o`
- Toutes les dépendances sont présentes : on peut donc compiler `exec`

Exécuter une cible en particulier

Par défaut, la cible exécutée est `all`

- Entrée dans le Makefile = première cible exécutée

```
1 all : exec
```

- Ici : la cible `all` dépend de `exec`, donc on va exécuter la cible `exec`

On peut exécuter une cible en particulier en passant son nom dans la ligne de commande de l'appel à `make` :

```
1 $ make nomCible
```

Cible particulière : clean

Une cible particulière permet de faire le ménage : on l'appelle généralement `clean`

```
1 clean :
2         rm -f *o exec
```

Ici :

- Elle n'a pas de dépendance : on effectue l'action directement
- Elle supprime les fichiers `*.o` et l'exécutable `exec`
- Elle sert à faire le ménage et supprimant les résultats de la compilation

Pour l'appeler :

```
1 $ make clean
```

Utilisation de variables

On peut déclarer des variables dans le Makefile

- Flexibilité du Makefile : possibilité de modifier des options à un seul endroit
- Définition de chemins, de compilateur à utiliser, d'options...

Utilisation : nom de la variable précédé de $, éventuellement parenthèses

- Définition du compilateur avec `CC`
- Utilisation avec `$(CC)`

```
1  CC = gcc
2  OPT = -O1 -g -Wall
3
4  all: programme
5
6  programme: programme.o outils.o programme.h
7          $(CC) $(OPT) -o programme programme.o outils.o
8
9  programme.o: programme.c programme.h
10         $(CC) $(OPT) -c programme.c
11
12 outils.o: outils.c programme.h
13         $(CC) $(OPT) -c outils.c
14
15 clean:
16         rm *.o programme
```