Modularité

François Trahay

Contents

Objectifs de la séance

- Savoir modulariser un programme

- Savoir définir une chaîne de compilation
- Maîtriser les options de compilations courantes

Modularité en C vs. Java

Beaucoup de concepts sont les même qu'en Java

- **Interface** d'un module: partie publique accessible d'autres modules
  - prototype des fonctions
  - définition des constantes et types
- **Implémentation** d'un module: partie privée
  - définition et initialisation des variables
  - implémentation des fonctions
- **Bibliothèque** (en anglais: *library*) : regroupe un ensemble de modules
  - Equivalent des *packages* java

Module en C

Deux fichiers par module. Par exemple, pour le module `mem_alloc`:

- Interface: fichier `mem_alloc.h` (fichier d'*entête* / *header*)
  - défini les constantes/types
  - déclare les prototypes des fonctions "publiques" (*ie.* accessible par les autres modules)
- Implémentation: fichier `mem_alloc.c`
  - utilise `mem_alloc.h` : `#include "mem_alloc.h"`
  - utilise les constantes/types de `mem_alloc.h`
  - déclare/initialise les variables
  - implémente les fonctions
- Utiliser le module `mem_alloc` (depuis le module `main`)
  - utilise `mem_alloc.h` : `#include "mem_alloc.h"`
  - utilise les constantes/types de `mem_alloc.h`
  - appelle les fonctions du module `mem_alloc`

Exemple: le module `mem_alloc`

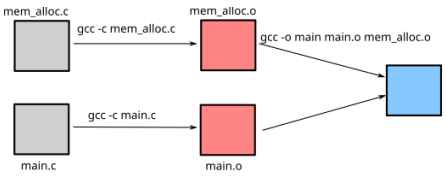
```c
/* mem_alloc.h */
#include <stdlib.h>
#include <stdint.h>
```

```c
#define DEFAULT_SIZE 16

typedef int64_t mem_page_t;
struct mem_alloc_t {
  /* [...] */
};

/* Initialize the allocator */
void mem_init();

/* Allocate size consecutive bytes */
int mem_allocate(size_t size);

/* Free an allocated buffer */
void mem_free(int addr, size_t size);


/* mem_alloc.c */
#include "mem_alloc.h"
struct mem_alloc_t m;
void mem_init() {  /* ... */  }
int mem_allocate(size_t size) {
  /* ... */
}
void mem_free(int addr, size_t size) {
  /* ... */
}


#include "mem_alloc.h"
int main(int argc, char**argv) {
  mem_init();
  /* ... */
}
```

Compilation de modules

Compilation en trois phases:

- Le **preprocesseur** prépare le code source pour la compilation
- Le **compilateur** transforme des instructions C en instructions “binaires”
- L’**éditeur de liens** regroupe des fichiers objets et crée un `exécutable`



Préprocesseur

Le **préprocesseur** transforme le code source pour le compilateur

- génère du code source
- interprète un ensemble de directives commençant par #
  - `#define N 12` substitue `N` par `12` dans le fichier
  - `#if <cond> ... #else ... #endif` permet la compilation conditionnelle
  - `#ifdef <var> ... #else ... #endif` (ou l'inverse: `#ifndef`) permet de ne compiler que si `var` est défini (avec `#define`)
  - `#include "fichier.h"` inclue (récursivement) le fichier `"fichier.h"`
- résultat visible avec : `gcc -E mem_alloc.c`

La directive `#if` permet, par exemple, de fournir plusieurs implémentations d'une fonction. Cela peut être utilisé pour des questions de portabilité.

```
#if __x86_64__
  void foo() { /* implementation pour CPU intel 64 bits */ }
#elif __arm__   /* équivalent à #else #if ... */
  void foo() { /* implementation pour CPU ARM */ }
#else
  void foo() {
    printf("Architecture non supportée\n");
    abort();
#endif
```

Il y a deux syntaxes pour la directive `#include`: * `#include <fichier>`: le préprocesseur cherche `fichier` dans un ensemble de répertoires systèmes (`/usr/include` par exemple) * `#include "fichier"" le préprocesseur cherche`fichier‘ dans le répertoire courant, puis dans les répertoires systèmes.

On utilise donc généralement `#include "fichier"` pour inclure les fichiers d'entête définis par le programme, et `#include <fichier>` pour les fichiers d'entête du système (`stdio.h`, `stdlib.h`, etc.)

Compilateur

Compilation : transformation des instructions C en instructions “binaires”

- appliquée à chaque module
- `gcc -c mem_alloc.c`
- génère le **fichier objet** `mem_alloc.o`
- génère des instructions "binaires" dépendantes du processeur

---

Editeur de liens

Edition de liens : regroupement des fichiers **objets** pour créer un **exécutable**

```
gcc -o executable mem_alloc.o module2.o [...] moduleN.o
```

Règles de compilations

- En cas de modification du corps d'un module (par exemple `mem_alloc.c`, il est nécessaire de regénérer le fichier objet (`mem_alloc.o`), et de regénérer l'exécutable. Il n'est toutefois pas nécessaire de recompiler les modules utilisant le module modifié.
- En cas de modification de l'interface d'un module (par exemple `mem_alloc.h`), il est nécessaire de recompiler le module, ainsi que tous les modules utilisant le module modifié. Une fois que tous les fichiers objets (les fichiers `*.o`) concernés ont été regénérés, il faut refaire l'édition de liens.

Lorsque le nombre de module devient élevé, il devient difficile de savoir quel(s) module(s) recompiler. On automatise alors la chaîne de compilation, en utilisant l'outil `make`.

Puisque la compilation se fait en 3 phases, 3 types d'erreurs peuvent survenir: * une erreur du préprocesseur (ie. une macro est mal écrite):

```
$ gcc -c foo.c
foo.c:1:8: error: no macro name given in #define directive
 #define
        ^
```

- une erreur lors de la compilation (ie. le programme est mal écrit):

```
  $ gcc -c erreur_compil.c
erreur_compil.c: In function ‘f’:
erreur_compil.c:3:1: error: expected ‘;’ before ‘}’ token
 }
 ^
```

- une erreur lors de l'édition de liens (ie. il manque des morceaux):

```
$ gcc -o plip main.o
main.o : Dans la fonction « main » :
main.c:(.text+0x15) : référence indéfinie vers « mem_init »
collect2: error: ld returned 1 exit status
```

Fichiers ELF

- Les fichiers objets et exécutables sont sous le format **ELF** (*Executable and Linkable Format*)
- Ensemble de sections regroupant les symboles d'un même type:
  - `.text` contient les fonctions de l'objet
  - `.data` et `.bss` contiennent les données initialisées (`.data`) ou pas (`.bss`)
  - `.symtab` contient la *table des symboles*
- Lors de l'édition de liens ou du chargement en mémoire, les sections de tous les objets sont fusionnés

La liste complète des sections du format ELF est disponible dans la documentation (`man 5 elf`).

La *table des symboles* contient la liste des fonctions/variables globales (ou statiques) définies ou utilisées dans le fichier. L'outil `nm` permet de consulter cette table. Par exemple:

```
$ nm mem_alloc.o
0000000000000000 C m
0000000000000007 T mem_allocate
0000000000000012 T mem_free
0000000000000000 T mem_init
$ nm main.o
0000000000000000 T main
                 U mem_init
```

Pour chaque symbole, `nm` affiche l'adresse (au sein d'une section), le type (donc, la section ELF), et le nom du symbole.

Ces informations sont également disponible via la commande `readelf`:

```
$ readelf  -s mem_alloc.o

Table de symboles « .symtab » contient 12 entrées :
   Num:    Valeur          Tail Type    Lien   Vis      Ndx Nom
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS mem_alloc.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    2
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
     5: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
     6: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
     7: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
```

```
     8: 0000000000000001     0 OBJECT  GLOBAL DEFAULT  COM m
     9: 0000000000000000     7 FUNC    GLOBAL DEFAULT    1 mem_init
    10: 0000000000000007    11 FUNC    GLOBAL DEFAULT    1 mem_allocate
    11: 0000000000000012    14 FUNC    GLOBAL DEFAULT    1 mem_free
```

L'utilitaire `objdump` permet lui aussi d'examiner la table des symboles:

```
$ objdump -t mem_alloc.o

mem_alloc.o:     format de fichier elf64-x86-64

SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 mem_alloc.c
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l    d  .note.GNU-stack    0000000000000000 .note.GNU-stack
0000000000000000 l    d  .eh_frame  0000000000000000 .eh_frame
0000000000000000 l    d  .comment   0000000000000000 .comment
0000000000000000       0 *COM*  0000000000000001 m
0000000000000000 g     F .text  0000000000000007 mem_init
0000000000000007 g     F .text  000000000000000b mem_allocate
0000000000000012 g     F .text  000000000000000e mem_free
```

Portée des variables locales

Une variable déclarée dans une fonction peut être

- **locale** : la variable est allouée à l'entrée de la fonction et désallouée à sa sortie
  - exemple: `int var;` ou `int var2 = 17;`
- **statique** : la variable est allouée à l'initialisation du programme.
  - Sa valeur est conservée d'un appel de la fonction à l'autre.
  - utilisation du mot-clé `static`
  - exemple: `static int var = 0;`

Puisqu'une variable locale statique est allouée au chargement du programme, elle apparaît dans la liste des symboles :

```
$ nm plop.o
0000000000000000 T function
0000000000000000 d variable_locale_static.1764
```

Ici, le symbole `variable_locale_static.1764` correspond à la variable `variable_locale_static` déclarée `static` dans la fonction `function`. Le suffixe `.1764` permet de différencier les variables nommées `variable_locale_static` déclarées dans des fonctions différentes.

Portée des variables globales

Une variable déclarée dans le fichier `fic.c` en dehors d'une fonction peut être:

- **globale** : la variable est allouée au chargement du programme et désallouée à sa terminaison
  - exemple: `int var;` ou `int var2 = 17;`
  - la variable est utilisable depuis d'autres objets
- **extern** : la variable est seulement déclarée
  - équivalent du prototype d'une fonction: la déclaration indique le type de la variable, mais celle ci est allouée (ie. déclarée globale) ailleurs
  - utilisation du mot-clé `extern`
  - exemple: `extern int var;`
- **statique** : il s'agit d'une variable globale accessible seulement depuis `fic.c`
  - utilisation du mot-clé `static`
  - exemple: `static int var = 0;`

Les variables globales (déclarées `extern`, `static`, ou "normales") se retrouvent dans la table des symboles de l'objet, mais dans des sections ELF différentes :

```
$ nm plop.o
0000000000000000 T function
                 U var_extern
0000000000000000 D var_globale
0000000000000004 d var_static_globale
```

La variable `var_extern` (déclarée avec `extern int var_extern;`) est marquée `"U"` (*undefined*). Il s'agit donc d'une référence à un symbole présent dans un autre objet.

La variable `var_globale` (déclarée avec `int  var_globale = 12;`) est marquée `"D"` (*The symbol is in the initialized data section*). Il s'agit donc d'une variable globale initialisée [1].

La variable `var_static_globale` (déclarée avec `static int var_static_globale = 7;`) est marquée "`d`" (*The symbol is in the initialized data section*). Il s'agit donc d'une variable globale "interne". Il n'est donc pas possible d'accèder à cette variable depuis un autre objet:

```
$ gcc plop.o  plip.o  -o executable
plip.o : Dans la fonction « main » :
plip.c:(.text+0xa) : référence indéfinie vers « var_static_globale »
collect2: error: ld returned 1 exit status
```

[1] D'après la documentation de nm à propos du type de symbole : "*If lowercase, the symbol is usually local; if uppercase, the symbol is global (external)*".

Bibliothèque

- Regroupement de fichiers objets au sein d'une bibliothèque
  - Equivalent d'un *package* Java
  - Accès à tout un ensemble de modules
- Utilisation
  - dans le code source: `#include "mem_alloc.h"`, puis utilisation des fonctions
  - lors de l'édition de lien: ajouter l'option `-l`, par exemple: `-lmemory`
    - Utilise la bibliothèque `libmemory.so` ou `libmemory.a`

Avantages/inconvénients des bibliothèques statiques:

- Taille de l'exécutable important (puisqu'il inclut la bibliothèque);

- En cas de nouvelle version d'une bibliothèque (qui corrige un bug par exemple), il faut recompiler toutes les applications utilisant la bibliothèque;

- Duplication du code en mémoire;

+ L'exécutable incluant une bibliothèque statique fonctionne "tout seul" pas besoin d'autres fichiers).

Avantages/inconvénients des bibliothèques dynamiques:

+ Taille de l'exécutable réduite (puisqu'il n'inclut qu'une référence à la bibliothèque);

+ En cas de nouvelle version d'une bibliothèque (qui corrige un bug par exemple), pas besoin de recompiler les applications utilisant la bibliothèque;

+ Une instance du code en mémoire est partageable par plusieurs processus;

- L'exécutable incluant une bibliothèque dynamique ne fonctionne pas "tout seul": il faut trouver toutes les bibliothèques dynamiques nécessaires.

Les "dépendances" dues aux bibliothèques dynamiques sont visibles avec `ldd`:

```
$ ldd executable
    linux-vdso.so.1 (0x00007fff9fdf6000)
    libmem_alloc.so (0x00007fb97cb9f000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fb97c7ca000)
    /lib64/ld-linux-x86-64.so.2 (0x0000555763a9b000)
```

Création d'une bibliothèque

Il existe 2 types de bibliothèques

- **Bibliothèque statique** : `libmemory.**a**`
  - Intégration des objets de la bibliothèque au moment de l'édition de liens

  - Création: `ar rcs libmemory.a mem_alloc.o mem_plip.o mem_plop.o [...]`
- **Bibliothèque dynamique** : `libmemory.**so**`
  - Intégration des objets de la bibliothèque à l'exécution
  - Lors de l'édition de liens: une référence vers la bibliothèque est intégrée à l'exécutable
  - Création: `gcc -shared -o libmemory.so mem_alloc.o mem_plip.o mem_plop.o [...]`
    * les objets doivent être créés avec l'option `-fPIC`:
    * `gcc -c mem_alloc.c -fPIC`

LD_LIBRARY_PATH

Pour exécuter un programme utilisant une bibliothèque dynamique, le système doit charger en mémoire le programme ainsi que la biblbiothèque. Si la bibliothèque n'est pas installée dans un répertoire standard (typiquement dans `/usr/lib`), il peut être nécessaire d'indiquer où trouver cette bibliothèque grâce à la variable d'environnement `LD_LIBRARY_PATH`.

```
$ ./mon_programme
mon_programme: error while loading shared libraries: libtruc.so: cannot open shared object f

$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/trahay/libs/libtruc/

$ ./mon_programme

It works !
```

Avantages et inconvénients

L'avantage d'une bibliothèque statique est qu'il n'est nécessaire de connaître son emplacement qu'au moment d'édition de liens. Une fois l'exécutable crée, celui-ci inclue la bibliothèque statique. On peut donc déplacer l'exécutable, supprimer la bibliothèque statique, recopier l'exécutable sur une autre machine sans empêcher son exécution.

Toutefois, lorsqu'une bibliothèque statique est mise à jour (par exemple pour corriger un bug ou une faille de sécutité), il est nécessaire de recompiler tous les programmes utilisant cette bibliothèque. Pour des bibliothèques très utilisées (par exemple, la libc), cela peut être long et on risque d'oublier de recompiler certains programmes.

A l'inverse, si une bibliothèque dynamique est mise à jour, cette mise à jour est directement disponible pour toutes les applications utilisant la bibliothèque. Ceci explique pourquoi la plupart des distributions Linux reposent aujourd'hui sur des bibliothèques dynamiques plutot que statiques.

Changement d'ABI

Attention, si une bibliothèque dynamique est mise à jour et que son ABI (Application Binary Interface) change (par exemple si la signature d'une fonction change), il sera nécessaire de recompiler les applications utilisant cette biblliothèque.

Organisation

Organisation classique d'un projet:

- `src/`
  - `module1/`
    - `module1.c`
    - `module1.h`
    - `module1_plop.c`
  - `module2/`
    - `module2.c`
    - `module2.h`
    - `module2_plip.c`
  - etc.
- `doc/`
  - `manual.tex`
- etc.

Besoin d'utiliser des `flags`:

- `-Idir` indique où chercher des fichiers `.h`

```
gcc -c main.c -I../memory/
```

- `-Ldir` indique à l'éditeur de lien où trouver des bibliothèques

```
gcc -o executable main.o -L../memory/ -lmem_alloc
```

A l'exécution:

- la variable `LD_LIBRARY_PATH` contient les répertoires où chercher les fichiers `.so`

```
export LD_LIBRARY_PATH=.:../memory
```

Par défaut, le compilateur va chercher les fichiers d'entête dans un certain nombre de répertoires. Par exemple, `gcc` cherche dans:

- `/usr/local/include`
- `<libdir>/gcc/<target>/version/include`
- `/usr/<target>/include`
- `/usr/include`

L'option `-I` ajoute un répertoire à la liste des répertoires à consulter. Vous pouvez donc utiliser plusieurs fois l'option `-I` dans une seule commande. Par exemple:

```
gcc -c main.o -Imemory/ -Itools/ -I../plop/
```

De même, l'éditeur de liens va chercher les bibliothèques dans un certain nombre de répertoires par défaut. La liste des répertoires parcourus par défaut par `ld` (l'éditeur de lien utilisé par `gcc`) est dans le fichier `/etc/ld.so.conf`. On y trouve généralement (entre autre):

- `/lib/<target>`
- `/usr/lib/<target>`
- `/usr/local/lib`
- `/usr/lib`
- `/lib32`
- `/usr/lib32`

Si la variable `LD_LIBRARY_PATH` est mal positionnée, vous risquez de tomber sur ce type d'erreur au lancement de l'application:

```
  $ ./executable
./executable: error while loading shared libraries: libmem_alloc.so: cannot open \
 shared object file: No such file or directory
```

Makefile

- Arbre des dépendances
  - "Pour créer `executable`, j'ai besoin de `mem_alloc.o` et `main.o`"
- Action
  - "Pour créer `executable`, il faut lancer la commande `gcc -o executable mem_alloc.o main.o`"
- Syntaxe: dans un fichier `Makefile`, ensemble de règles sur deux lignes:

```
cible : dependance1 dependance2 ... dependanceN`
<TAB>commande
```

- Pour lancer la compilation: commande `make`
  - Parcourt l'arbre de dépendance et détecte les cibles à regénérer
  - Exécute les commandes pour chaque cible

La commande `make`

- La commande `make` parcourt le fichier `Makefile` du répertoire courant et tente de produire la première *cible*.
- Il est également possible de spécifier une cible à produire en utilisant `make cible`.

- Dès qu'une *action* génère une erreur, la commande `make` s'arrête
- La liste des cibles à regénérer est calculée à partir de l'arbre de dépendance décrit dans le fichier `Makefile` et des date/heure de dernière modification des fichiers: si un fichier de l'arbre est plus récent que la cible à générer, tout le chemin entre le fichier modifié et la cible est regénéré.

Le fichier `Makefile`

Voici un exemple de fichier `Makefile`:

```
all: executable

executable: mem_alloc.o main.o
        gcc -o executable main.o mem_alloc.o

mem_alloc.o: mem_alloc.c mem_alloc.h
        gcc -c mem_alloc.c

main.o: main.c mem_alloc.h
        gcc -c main.c
```

- La (ou les) commande(s) à exécuter pour générer chaque cible est précédée du caractère *Tabulation*. Sous `emacs`, lorsque vous éditez un fichier nommé `Makefile`, les tabulations sont colorisées en rose par défaut.

Si vous utilisez des espaces à la place de la tabulation, la commande `make` affiche le message d'erreur suivant:

```
Makefile:10: *** missing separator (did you mean TAB instead of 8 spaces?). Arrêt.
```

- Puisque la commande `make` (sans argument) génère la première cible, on ajoute généralement une premier cible "artificielle" `all` décrivant l'ensemble des exécutables à générer:

```
all: executable1 executable2
```

Dans ce cas, seule la cible est spécifiée. Il n'y a pas d'action à effectuer.

Règle `clean`

- On ajoute également souvent une règle artificielle `clean` pour "faire le ménage" (supprimer les exécutables et les fichiers `.o`):

```
clean:
<TAB>rm -f executable1 executable2 *.o
```

- Lorsqu'on écrit un fichier `Makefile`, on peut utiliser certaines notations symboliques:
  - `$@` désigne la cible de la règle

  - `$^` désigne l'ensemble des dépendances
  - `$<` désigne la première dépendance de la liste
  - `$?` désigne l'ensemble des dépendances plus récentes que la cible
- Il est possible de définir et d'utiliser des variables dans un fichier `Makefile`. Par exemple:

```
BIN=executable
OBJETS=mem_alloc.o main.o
CFLAGS=-Wall -Werror -g
LDFLAGS=-lm

all: $(BIN)

executable: $(OBJETS)
	gcc -o executable main.o mem_alloc.o $(LDFLAGS)

mem_alloc.o: mem_alloc.c mem_alloc.h
	gcc -c mem_alloc.c $(CFLAGS)

main.o: main.c mem_alloc.h
	gcc -c main.c $(CFLAGS)

clean:
	rm -f $(BIN) $(OBJETS)
```

Configuration et dépendances

- Dans "la vraie vie", l'outil `make` n'est qu'une partie de la chaîne de configuration et de compilation.
- Des outils comme `autoconf`/`automake` ou `Cmake` sont fréquemment utilisés pour écrire des "proto-makefiles"
- Ces outils permettent de détecter la configuration de la machine (quel CPU ? la bibliothèque X est-elle installée ? etc.) et de définir les options de compilation de manière portable.