

Modularité

François Trahay



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)
 - définit 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

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

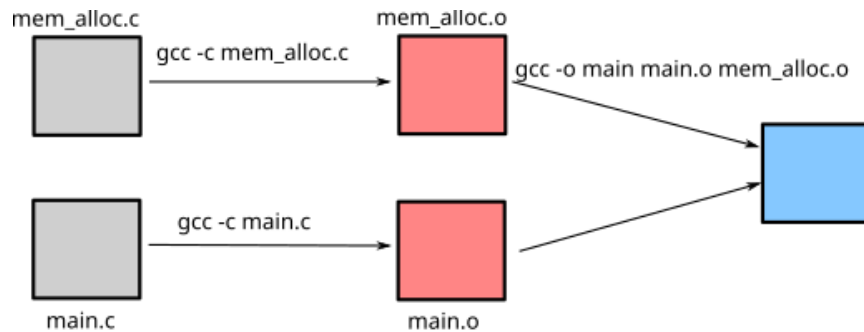
```
/* mem_alloc.c */
#include "mem_alloc.h"
struct mem_alloc_t m;
void mem_init() { /* ... */ }
int mem_allocate(size_t size) {
    /* ... */
}
void mem_free(int addr, size_t size) {
    /* ... */
}
```

```
#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`

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
```

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

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;`

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;`

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`

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`

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:

- -I*dir* indique où chercher des fichiers .h

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

- -L*dir* 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
```

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 régénérer
 - Exécute les commandes pour chaque cible