Programmation « orientée système »
Langage C – Compilation (2/2)

Jean-Cédric Chappelier

Laboratoire d'Intelligence Artificielle
Faculté I&C

Objectifs du cours d'aujourd'hui

- Compilation séparée
-  Makefile 

Approche modulaire

Jusqu'à maintenant vos programmes étaient écrits en une seule fois, dans un seul fichier.

Cette approche n'est pas réaliste pour des programmes plus conséquents, qui nécessitent partage de composants, maintenance séparée, réutilisation, ...

On préfère une **approche modulaire**, c'est-à-dire une approche qui

décompose la tâche à résoudre en sous-tâches

implémentées sous la forme de **modules génériques** (qui pourront être **réutilisés** dans d'autres contextes).

Chaque module correspond alors à une tâche ponctuelle, à un ensemble cohérent de données, à un concept de base, etc.

Jean-Cédric Chappelier

Utilité

☞ Pourquoi faire cela ?

- Pour rendre **réutilisable** : éviter de réinventer la roue à chaque fois

  La conception d'un programme doit tenir compte de deux aspects importants :
  - la **réutilisation** des objets/fonctions existants : bibliothèques logicielles (« libraries » en anglais) ;
    (les autres/passé $\longrightarrow$ nous/présent)
  - la **réutilisabilité** des objets/fonctions nouvellement créés.
    (nous/présent $\longrightarrow$ les autres/futur)
- Pour **maintenir** plus **facilement** : pas besoin de tout recompiler le jour où on corrige une erreur dans une (sous-...-sous-)fonction
- Pour pouvoir **développer** des programmes **indépendamment**, c'est-à-dire même si le code source n'est pas disponible
- **Distribuer des bibliothèques** logicielles (morceaux de code) sans en donner les codes sources (protection intellectuelle).
  Remarque : vous pouvez vous-même créer vos propres bibliothèques.

Jean-Cédric Chappelier
EPFL

Conception modulaire

Concrètement, cela signifie que les types, structures de données et fonctions correspondant à un « concept de base » seront **regroupés dans un fichier** qui leur est propre.

Par exemple, on définira la structure `qcm` et ses fonctions dans un fichier, à part de son utilisation.

☞ séparation des déclarations des objets de leur utilisation effective (dans un `main()`).

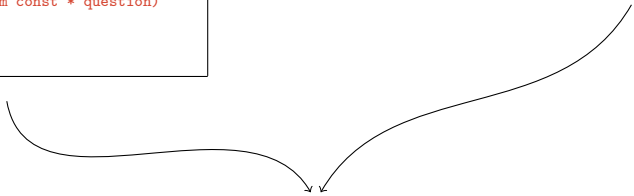
Concrètement, cela crée donc plusieurs fichiers séparés qu'il faudra regrouper (« lier ») en un tout pour faire un programme.

Jean-Cédric Chappelier

Exemple : exercice sur les QCM

```
typedef struct { ... } qcm;

void affiche(qcm const * question);
int poser_question(qcm const * question);
...

void affiche(qcm const * question)
{
   ...
}

int poser_question(qcm const * question)
{
   ...
}
```

```
int demander_nombre(int min, int max);

int demander_nombre(int a, int b) {
  ...
}
```



```
int main(void)
{
   qcm maquestion;

   ...
   poser_question(maquestion);
}
```

Compilation séparée

Le but est de séparer chacun de ces « concepts » dans un fichier séparé.

☞ Mais comment alors faire un tout (un programme complet) ?
Comment `main()` connait-il le reste ?
Comment les QCMs connaissent-ils `demander_nombre()` ?

La partie déclaration est la partie **visible** du module que l'on écrit, qui va permettre son utilisation (et donc sa réutilisation).

C'est elle qui est utile aux autres fichiers pour utiliser les objets déclarés.

La partie définition est l'implémentation du code correspondant et n'est pas directement nécessaire pour l'utilisateur du module.
Elle peut être **cachée** (aux autres).

Jean-Cédric Chappelier
EPFL

Compilation séparée

De ce fait, il est nécessaire (en conception modulaire) de séparer *chacune* de ces parties, en deux fichiers :

- les fichiers de déclaration (fichiers « *headers* »), avec une extension `.h`.

  Ce sont ces fichiers qu'on inclut en début de programme par la commande `#include`

- les fichiers de définition (fichiers sources, avec une extension `.c`)

  Ce sont ces fichiers que l'on compile pour créer du code exécutable.

A quoi sert donc un fichier `.h` ?

☞ A ce que les *autres* fichiers `.c`, qui utilisent ce module, puissent compiler.

Règles :

1. Pour chaque fichier (`.c` ou `.h`), pris/considéré *indépendemment* (= « pour lui-même ») : y mettre **tous** les `#include` dont *ce* fichier a besoin et **uniquement** ceux dont *il* a besoin ! Ni plus, ni moins !
2. Faire commencer le fichier `.h` par « `#pragma once;` » afin d'éviter les inclusions multiples.

Compilation séparée : exemple

```c
typedef struct { ... } qcm;

void affiche(qcm const * question);
int poser_question(qcm const * question);
...

void affiche(qcm const * question)
{
   ...
}

int poser_question(qcm const * question)
{
   ...
}
```

```c
#include "qcm.h"
void affiche(qcm const * question)
{
   ...
}

int poser_question(qcm const * question)
{
   ...
}
```

qcm.c

```c
typedef struct { ... } qcm;

void affiche(qcm const * question);
int poser_question(qcm const * question);
...
```

qcm.h

Jean-Cédric Chappelier

Compilation séparée (2)

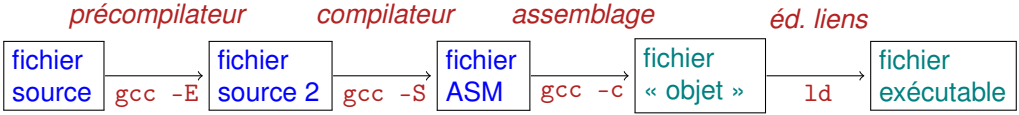
La séparation des parties déclaration et définition en deux fichiers permet une **compilation séparée** du programme complet :

- phase 1 (compilation) : production de fichiers binaires (appelés **fichiers objets**) correspondant à la compilation des fichiers sources (`.c`) contenant les parties définitions (et dans lesquels on inclut (`#include`) les fichiers « *headers* » (`.h`) nécessaires) ;
- phase 2 (édition de liens) : production du fichier exécutable final à partir des fichiers objets et des éventuelles bibliothèques.

Note : pour un programme en *N* parties (`.c`), on fait *N* fois la phase de compilation et 1 seule fois la phase d'édition de liens.

Jean-Cédric Chappelier

Compilation d'un programme C



hello_E.c

```
typedef long unsigned int size_t;

typedef unsigned char __u_char;
typedef unsigned short int __u_short;
typedef unsigned int __u_int;
typedef unsigned long int __u_long;

typedef signed char __int8_t;
typedef unsigned char __uint8_t;
typedef signed short int __int16_t;
typedef unsigned short int __uint16_t;
typedef signed int __int32_t;
...
```

hello.asm

```
        .file   "hello.c"
        .section .rodata
.LC0:   .string "Hello World!"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:  .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $.LC0, %edi
...
```

Compilation *séparée* d'un programme C

```
gcc -c qcm.c -o qcm.o
```

fichier source1 → fichier objet1

compilateur

fichier source2 → fichier objet2

```
gcc -c questionnaire.c -o questionnaire.o
```

édition de liens

```
gcc questionnaire.o qcm.o -o questionnaire
```

fichier exécutable

En fait, `gcc` appelle ici l'éditeur de lien `ld`

Jean-Cédric Chappelier
EPFL

Rôle de l'édition de liens

Les différentes composantes d'un programme ayant été compilées séparément, le code compilé (ou « code objet ») contient des *références* à des bouts de codes *non connus* au moment la compilation.

(c'est aussi vrai pour un programme contenu dans un seul fichier : il utilise toujours des bibliothèques du systèmes [ne serait-ce que la `libc` !] qui ont été compilées [bien] avant lui !)

Le rôle de l'édition de liens (« linker ») est précisément de construire ces liens entre bouts de codes compilés séparément : résoudre les ambiguïtés d'appel

Rôle de l'édition de liens (2)

Un code objet, c'est en fait du code partiel + des tables d'adressage

Il contient trois types de tables :

- table d'**exportation** des objets globaux (variables ou fonctions) ;
- table d'**importation** des objets référencés, mais d'adresse inconnue ;
- table des **tâches** : liste des endroits dans le code où se trouvent les adresses à résoudre.

Jean-Cédric Chappelier
EPFL

Rôle du chargeur

Mais même l'édition de liens ne peut pas tout résoudre...

Au moment de son « chargement » (loading) pour exécution par le système d'exploitation, restent encore dans le programme certains détails d'adresses locales à régler.

C'est précisément le rôle du **chargeur** (« loader »).

Le chargeur est un module du système d'exploitation dont le rôle est de résoudre les dernières ambiguïtés liées au placement effectif en mémoire du programme exécutable avant de lancer son exécution proprement dite.

En pratique, contrairement au compilateur et à l'éditeur de liens, vous ne voyez pas explicitement ce module.

Jean-Cédric Chappelier

Linker/Loader : exemple

Reprenons notre exemple de `QCM`.

Lors de la compilation du programme principal `questionnaire.c`, le compilateur ne connaît pas l'adresse mémoire du code correspondant aux fonctions déclarées dans `qcm.h`

☞ le compilateur laisse cette partie du travail (résoudre les adresses inconnues) au linker

Tables d'exportation :

« code » ou « variable »

dans `qcm.o` :

nom	type	adresse
`affiche`	code	0 en relatif
`poser_question`	code	342 en relatif (adresse de la première instruction de cette fonction par rapport à tout le code de ce module)

dans `questionnaire.o` :

nom	type	adresse
`main`	code	0 en relatif

Linker/Loader : exemple (suite)

Table d'importation :

`qcm.c` n'en a pas (tous les objets qui y sont référencés sont connus)

pour `questionnaire.c` : `affiche`, `poser_question`, et peut être aussi `sqrt` (ou autres fonctions de bibliothèques système)

Table des tâches :

pour `qcm.c` : tous les sauts en mémoire (par exemple dus à des structures de contrôle).

pour `questionnaire.c` : idem `qcm.c`, plus tous les endroits où un appel à du code importé existe.

Dans ce cas, les valeurs à résoudre sont exprimées en termes d'entrées dans la *table d'importation*, lesquelles seront résolues lors de l'**édition de lien** par consultation des *tables d'exportation* des autres codes objets.

Pour finir, le **chargeur** modifie toutes les adresses de saut en fonction de l'adresse de chargement du programme (point d'entrée)

.

(on dit que le chargeur « translate » le code)

Compilation conditionnelle
Exemple utile pour la compilation séparée

Les variables globales définies dans un module et utilisées dans un autre doivent :

- être définies uniquement dans le `.c`
  (si elles étaient dans le `.h`, il y aurait ambiguïté (par duplication) puisqu'elles seraient présentes dans chaque module qui `#include` ce fichier `.h`)
- être déclarées dans les modules qui les utilisent
  ...donc mises dans le `.h` !
  Pour ces modules là, elles doivent être déclarées comme
  `extern`

MAIS *si* ce fichier `.h` *doit* être inclus dans son fichier `.c` correspondant (p.ex. parce qu'il contient aussi des définitions de types)...
...alors il *ne* faudrait *pas* que ces variables globales aient le mot `extern`...

Comment faire **sans** duplication de code ?...
☞ compilation conditionnelle

Jean-Cédric Chappelier
EPFL

Compilation conditionnelle et compilation séparée (2)

Supposons que l'on ait un identificateur unique du fichier `qcm.c` (disons `QMC_C`), on pourrait alors écrire :

```
// JAMAIS de variable globale !!

#ifndef QCM_C
extern
#endif
int nombre_questions
#ifndef QCM_C
= 0
#endif
;
```

qcm.h

```
#define QCM_C

#include "qcm.h"

...
```

qcm.c

Jean-Cédric Chappelier

Compléments sur les headers files

Un même fichier `.h` pourrait se trouver inclus plusieurs fois dans la compilation via d'autre fichier `.h`.

Pour éviter des redéfinition multiples et garantir que le contenu d'un fichier `.h` n'est présent qu'une seule fois dans une compilation donnée, on utilise le truc suivant

- définition d'un identificateur «unique» au début du fichier
  Par convention c'est souvent le nom du projet suivit du nom du fichier en majuscules avec des _ à la place des caractères non alphanumériques.
- inclusion conditionnelle du fichier (y compris la définition ci-dessus)

Cela donne :

```
#ifndef MONFICHIERAMOI_H
#define MONFICHIERAMOI_H
   // ... le fichier comme d'habitude
#endif
```

On peut aussi utiliser : `#pragma once`

Jean-Cédric Chappelier

Compléments sur les headers files (2)

Les modules écrits en C peuvent également être utilisés en C++

Mais le C++ requiert que de telles fonctions soient déclarées en `extern "C"`

Pour faire un fichier d'en-tête C portable en C++, on utilisera donc une nouvelle fois la compilation conditionnelle comme suit :

```
#ifdef __cplusplus
extern  "C" {
#endif

   // ... le fichier comme d'habitude

#ifdef __cplusplus
}
#endif
```

Compléments sur les headers files (résumé)

Pour résumer, voici à quoi ressemble un fichier d'en-tête «bien» écrit
(il manque cependant encore de commentaires !) :

```
#pragma once

#ifdef __cplusplus
extern  "C" {
#endif

/* pas nécessaire pour les fonctions
 * mais obligatoire pour les variables (globales) */
#ifndef QCM_C
#define extern_ extern
#else
#define extern_
#endif

// prototypes fonctions et autres...

// exemple de variable globale (à éviter !!)
extern_ unsigned int nombre_qcm;

#undef extern_

#ifdef __cplusplus
}
#endif
```


Jean-Cédric Chappelier

Makefile (introduction)

Mais quand on a un grand nombre de modules, cela devient vite **fastidieux** de faire toutes ces compilations et ces liens...

...pour cela il y a des **moyens plus pratiques** dont les **Makefile**

Un `Makefile` est un fichier qui permet de construire facilement un projet en indiquant les composants et leurs dépendances.

(« `Makefile` » est vraiment le nom de ce fichier, sans extension `.qqchose` ;
c'est juste un fichier texte.)

Une fois un `Makefile` constitué, pour réaliser l'exécutable correspondant au projet il suffit de taper simplement `make`.

ou alors pour construire un programme particulier `cible` :
`make cible`.

EPFL

Makefile (bases)

Un `Makefile` a une structure très simple : il est constitué d'un ensemble de règles décrivant les différents modules à faire et de quoi ils dépendent (« *liste de dépendances* »).

Une règle s'écrit :

```
but: liste de dépendances
```

Exemple :

```
questionnaire: demander_nombre.o qcm.o questionnaire.o
```

La première règle écrite dans le fichier `Makefile` permet de donner la liste de tous les exécutables que l'on veut créer ; par exemple :

```
all: questionnaire
```

Si on a des bibliothèques système à utiliser, il faut les ajouter dans la variable `LDLIBS` au début du `Makefile` :

```
LDLIBS = -lm
```

Makefile (exemple simple)

Exemple (simple) complet :

```
LDLIBS = -lm

all: questionnaire

questionnaire: demander_nombre.o qcm.o questionnaire.o

questionnaire.o: questionnaire.c qcm.h
qcm.o: qcm.c qcm.h demander_nombre.h
```

Remarques :

1. On peut ajouter d'autres options au compilateur avec la variable `CFLAGS`.
   Par exemple : `CFLAGS += -g -std=c17`
2. On peut obtenir automatiquement les dépendances de compilation (c.-à-d. les dépendances des fichiers `.c`) à l'aide de la commande :
   `gcc -MM *.c`
3. `make` utilise des **règles implicites**.
   On peut donc exprimer encore beaucoup plus de choses dans un `Makefile`.

Makefile (suite)

On n'est pas obligé d'utiliser les règles implicites de compilation, mais on peut, au cas par cas, spécifier exactement la/les commandes que l'on souhaite exécuter pour passer des dépendances au but.
Cela se fait de la façon suivante

```
but: liste de dépendances
<TAB>commande
```

où <TAB> représente **une tabulation**
(j'insiste : pas 4 ou 8 espaces, mais 1 seul caractère <TAB> !)

Exemple :

```
questionnaire: questionnaire.o qcm.o
<TAB>gcc -o questionnaire questionnaire.o qcm.o
```

On peut définir plusieurs commandes à la suite pour une même cible. Il suffit de les mettre chacune à la ligne précédée d'un <TAB>

Elles sont alors exécutées par `make` les unes après les autres
(nouveau Shell à chaque fois)

Makefile : variables prédéfinies

Afin de faciliter l'écriture des commandes dans un `Makefile`, un certain nombre de variables sont prédéfinies

`$@`	le but
`$?`	les dépendances qui ne sont plus à jour
`$<`	dépendances telles que définies par les règles par défaut
`$^`	[GNU make] liste des dépendances
`$(CC)`	le nom du compilateur (C)
`$(CFLAGS)`	options de compilation
`$(LDFLAGS)`	options du linker
`$(LDLIBS)`	bibliothèques à ajouter

Exemples :

```
questionnaire.o: questionnaire.c qmc.h
<TAB>gcc -o $@ $<
```


Jean-Cédric Chappelier

Makefile : variables prédéfinies (2)

Exemple de règles par défaut exprimées avec les variables prédéfinies :

compilation `.c` → `.o` :

```
$(CC) -c $(CPPFLAGS) $(CFLAGS) $<
```

édition de liens :

```
$(CC) -o $@ $(LDFLAGS) $^ $(LDLIBS)
```


Jean-Cédric Chappelier

Makefile : variables

On peut également définir ses propres variables.

La déclaration se fait simplement avec le nom de la variable suivit de `=`

Exemple :

```
RUBS = *.o *~ *.bak
```

Pour utiliser la valeur d'une variable on entoure son nom de `$( )`

Exemple : `$(RM) $(RUBS)`

Les variables peuvent être redéfinies lors de l'appel :

```
make LDLIBS=-lm monprog
```

redéfinit la variable `LDLIBS`.

Jean-Cédric Chappelier

Makefile : divers (1/2)

- On peut mettre des commentaires dans un Makefile

  Tout ce qui suit derrière un `#` jusqu'à la fin de la ligne est considéré comme un commentaire

- Si l'on fait précéder la commande donnée dans une règle par `@` la commande n'est pas répétée à l'écran lors de l'exécution de `make` (c.-à-d. no echo)

- Si l'on fait précéder la commande donnée dans une règle par `-`, `make` continue l'exécution même en cas d'échec de cette commande

- On peut générer automatiquement la liste de toutes les dépendances en utilisant l'option `-MM` de `gcc` :

  ```
  gcc -MM *.c
  ```


Jean-Cédric Chappelier

Makefile : divers (2/2)

- Il existe plusieurs outils pour générer automatiquement les `Makefiles` en fonction de la configuration de la machine.

  Voir par exemple :
  - CMake, http://www.cmake.org,
  - SCons, http://http://www.scons.org/,
  - gyp, http://https://code.google.com/p/gyp/,
  - ninja, http://http://martine.github.io/ninja/,
  - Jam (BJam, KJam, ...),
  - the GNU Build Tools, alias « autotools » (`automake`, `autoconf` and `libtool`), cf http://sourceware.org/autobook/, http://autotoolset.sourceforge.net/tutorial.html
  - les outils intégrés de développement de projets (IDE) : Code : :Blocks, KDevelop, Anjuta, NetBeans, Eclipse, ...

Jean-Cédric Chappelier

Makefile : exemple

```
# Makefile pour le projet BIDULEMACHIN
#  cree par C. J. Reileppach le 11/03/2018

CC       = gcc
CFLAGS  += -std=c17 -g -Wall
LDLIBS  += -lm

TARGETS = bidulemachin
OBJS = *.o
RUBS = $(OBJS) *~ core \#*\#

all: $(TARGETS)
        @echo All done.

clean:
        -@$(RM) $(RUBS)
        @echo Cleaned.

new: clean
        -@$(RM) $(TARGETS)
        $(MAKE) all
```


Jean-Cédric Chappelier

Compulation séparée

Compilation modulaire
⇒ séparation des prototypes (dans les fichier `.h`) des définitions (dans les fichiers `.c`)

⇒ compilation séparée

1. Inclusion des prototypes nécessaires dans le code :
   `#include "header.h"`
2. Compilation vers un fichier "objet" (`.o`) : `gcc -c prog.c`
3. Lien entre plusieurs objets :
   `gcc prog1.o prog2.o prog3.o -o monprog`

`Makefile` :
moyen utile pour décrire les dépendances entre modules d'un projet (et compiler automatiquement le projet)

Syntaxe :

```
cible: dependance <TAB>commande
```


Jean-Cédric Chappelier