COURS 6 - COMPILATION SÉPARÉE, CLASSES DE STOCKAGE ET MAKEFILE

G. Bianchi, G. Blin, A. Bugeau, S. Gueorguieva, R. Uricaru

2015-2016

Programmation 1 - uf-info.ue.prog1@diff.u-bordeaux.fr

université de BORDEAUX

LA PROGRAMMATION MODULAIRE

▷ Trois grands principes à suivre

▷ L'utilisation de constantes symboliques définies au moyen de la directive `#define`

▷ La factorisation du code à l'aide de fonctions pour éviter la duplication de code

▷ La fragmentation du code en plusieurs fichiers afin d'augmenter la lisibilité et la réutilisation aisée de portions de code par d'autres applications
→ Compilation séparée

COMPILATION SÉPARÉE

PRINCIPE DE BASE

▷ Chaque fichier source contient un ensemble de fonctions généralement correspondant à un ”thème” (e.g., fonctions d’affichage)

▷ Les fichiers sources sont compilés séparément

▷ Les binaires produits seront liés lors de l’édition des liens

EXEMPLE

main.c

```c
#include <stdlib.h>
#include <stdio.h>
#define GOLDEN_RATIO 1.618
float daliGoldenRectangle(float x){
    return x*GOLDEN_RATIO;
}
int main(void){
    float a=1, b=0, c=0;
    b=daliGoldenRectangle(a);
    c=daliGoldenRectangle(a+b);
    printf("(%f,%f)\n",a,b);
    printf("(%f,%f)\n",a+b,c);
    return EXIT_SUCCESS;
}
```

gcc -std=c99 -Wall -c main.c
gcc -std=c99 -Wall -c dali.c
gcc -o example main.o dali.o

La déclaration de la fonction `daliGoldenRectangle` dans `main.c` est obligatoire ici

dali.c

```c
#define GOLDEN_RATIO 1.618
float daliGoldenRectangle(float x){
    return x*GOLDEN_RATIO;
}
```

main.c

```c
#include <stdlib.h>
#include <stdio.h>
float daliGoldenRectangle(float x);

int main(void){
    float a=1, b=0, c=0;
    b=daliGoldenRectangle(a);
    c=daliGoldenRectangle(a+b);
    printf("(%f,%f)\n",a,b);
    printf("(%f,%f)\n",a+b,c);
    return EXIT_SUCCESS;
}
```

Fichier en-tête d'un fichier source

▷ Pour que le programme reste modulaire, on place la déclaration de la fonction `daliGoldenRectangle` dans un fichier en-tête `dali.h` que l'on inclut dans `main.c` à l'aide de `#include`

▷ Ce fichier contient les déclarations des fonctions utilisables en dehors du fichier ainsi que les définitions des constantes symboliques et des macros utilisées

▷ Le fichier d'en-tête doit être inclus dans tous les fichiers sources utilisant une des fonctions définies dans le fichier source correspondant (e.g., `dali.c`) – y compris dans le fichier source lui-même

▷ L'inclusion du fichier d'en-tête (e.g., `dali.h`) dans le fichier source correspondant (e.g., `dali.c`) permet au compilateur de vérifier la compatibilité de la déclaration et de la définition de la fonction

EXEMPLE

main.c

```c
#include <stdlib.h>
#include <stdio.h>
#define GOLDEN_RATIO 1.618
float daliGoldenRectangle(float x){
    return x*GOLDEN_RATIO;
}
int main(void){
    float a=1, b=0, c=0;
    b=daliGoldenRectangle(a);
    c=daliGoldenRectangle(a+b);
    printf("(%f,%f)\n",a,b);
    printf("(%f,%f)\n",a+b,c);
    return EXIT_SUCCESS;
}
```

`#include <foo.h>` - recherche dans la librairie standard
`#include "foo.h"` - recherche dans le répertoire local

dali.h

```c
#define GOLDEN_RATIO 1.618
float daliGoldenRectangle(float x);
```

dali.c

```c
#include "dali.h"
float daliGoldenRectangle(float x){
    return x*GOLDEN_RATIO;
}
```

main.c

```c
#include <stdlib.h>
#include <stdio.h>
#include "dali.h"
int main(void){
    float a=1, b=0, c=0;
    b=daliGoldenRectangle(a);
    c=daliGoldenRectangle(a+b);
    printf("(%f,%f)\n",a,b);
    printf("(%f,%f)\n",a+b,c);
    return EXIT_SUCCESS;
}
```

FICHIER EN-TÊTE

▷ ⚠ Les fichiers d'en-tête ne doivent pas être compilés

▷ Il faut empêcher la possible double inclusion et donc les possibles définitions multiples d'un fichier d'en-tête

▷ A l'aide d'une constante symbolique, habituellement appelée `_NOM_H_` définie au début du fichier `nom.h` et dont l'existence est précédemment testée

  ▷ Si cette constante est définie alors c'est que le fichier en-tête `nom.h` a déjà été inclus

  ▷ Sinon, on définit la constante et on prend en compte le contenu du fichier `nom.h`

INCLUSION CONDITIONNELLE EN PRATIQUE

▷ `#ifdef NOM` permet de prendre en compte toutes les instructions jusqu'au `#else`, `#elif` ou `#endif` correspondant si et seulement si la macro **NOM** a été définie

▷ `#ifndef NOM` a un comportement identique dans le cas où la macro **NOM** n'a pas été définie

sum.h

```
#ifndef _DALI_H_
    #define _DALI_H_
    #define GOLDEN_RATIO 1.618
    float daliGoldenRectangle(float x);
#endif
```

CLASSES DE STOCKAGE

EXTERN

▷ Le mot clef **`extern`** permet de déclarer quelque chose sans le définir, ni réserver l'espace mémoire correspondant

▷ Pour une fonction, cela ne fait pas de différence mais montre explicitement qu'elle n'est pas définie dans le même fichier

```
#include <stdlib.h>
#include <stdio.h>
#define GOLDEN_RATIO 1.618

extern float daliGoldenRectangle(float x); //Better include headers

int main(void){
    float a=1, b=0, c=0;
    b=daliGoldenRectangle(a);
    c=daliGoldenRectangle(a+b);
    printf("(%f,%f)\n",a,b);
    printf("(%f,%f)\n",a+b,c);
    return EXIT_SUCCESS;
}
```

EXTERN

▷ Même si cela doit être évité, il est parfois nécessaire d'utiliser une variable commune à plusieurs fichiers sources

▷ Le compilateur doit comprendre que deux variables portant le même nom mais déclarées dans deux fichiers différents correspondent en fait à une seule variable commune

H2G2.c

```
int shared_variable=42;
void ultimate_question(){
    printf("Answer of the Ultimate Question of Life, the Universe \
    and Everything : %d\n", shared_variable);
}
```

Base13.c

```
int shared_variable;
void six_by_night(){
    printf("Six times Night in base 13  : %d\n", shared_variable);
}
```

EXTERN

▷ La variable doit être déclarée une et une seule fois de manière classique (réservation de l'espace mémoire correspondant)

▷ Dans les autres fichiers qui l'utilisent, il faut déclarer cette variable à l'aide du mot clef **`extern`** (cela ne donne pas lieu à une réservation d'espace mémoire)

H2G2.c

```
int shared_variable=42;
void ultimate_question(){
    printf("Answer of the Ultimate Question of Life, the Universe \
    and Everything : %d\n", shared_variable);
}
```

Base13.c

```
extern int shared_variable;
void six_by_night(){
    printf("Six times Night in base 13  : %d\n", shared_variable);
}
```

VARIABLE PARTAGÉE

▷ ⚠Même avec la protection par macros, si on met une variable dans un **.h**, des problèmes persistent

const.h

```
#ifndef _CONST_H_
    #define _CONST_H_
    extern void foo(void);
    int MODE=12;
#endif
```

foo.c

```
#include "const.h"
void foo(void) {
   MODE=MODE+5;
}
```

main.c

```
#include <stdio.h>
#include "const.h"
int main(int argc,char* argv[]) {
    foo();
    return 0;
}
```

```
$>gcc -Wall -std=c99 main.c foo.c
ccs3baaa.o(.data+0x0):foo.c: multiple definition of `MODE'
ccmqbaaa.o(.data+0x0):main.c: first defined here
```

VARIABLE PARTAGÉE

▷ Solution: mettre la variable dans un `.c` et sa déclaration externe dans le `.h`

const.h

```
#ifndef _CONST_H_
    #define _CONST_H_
    extern void foo(void);
    extern int MODE;
#endif
```

const.c

```
int MODE=42;
```

foo.c

```
#include "const.h"
void foo(void) {
   MODE=MODE+5;
}
```

main.c

```
#include <stdio.h>
#include "const.h"
int main(int argc,char* argv[]) {
    foo();
    return 0;
}
```

RESTRICTION DE LA VISIBILITÉ D'UNE VARIABLE

▷ Le mot clef **`static`** permet de ne pas rendre visible de l'extérieur une variable globale ou une fonction

foo.c

```
static int var1;
int var2;
static void foo1(void) {}
void foo2(void) {}
```

test.c

```
extern int var1, var2;
extern void foo1(void);
extern void foo2(void);

int main(int argc,char* argv[]) {
    foo1();
    foo2();
    return 0;
}
```

```
$>gcc -Wall -std=c99 test.c foo.c
foo.c:1: warning: 'var1' defined but not used
foo.c:4: warning: 'foo1' defined but not used
cc8qbaaa.o(.text+0x1f):test.c: undefined reference to 'foo1'
```

VARIABLE GLOCALE

▷ Une variable locale `static` correspond à une variable globale non visible de l'extérieur de la fonction

```
/**
* This function counts the number of
* function calls
*/
void count_calls(){
    static int count=0; // only initialized once
    count=count+1;
}
```

VARIABLE À ACCÈS RAPIDE

▷ Le mot clef **`register`** permet de demander à utiliser un registre pour une variable (dans la mesure du possible)

```
#define LIM 2000
int main(int argc,char* argv[]) {
    OPT unsigned int i, j, k, res;
    res=0;
    for (i=0;i<LIM;i++){
        for (j=0;j<LIM;j++){
            for (k=0;k<LIM;k++){
                res=res+i+j+k;
            }
        }
    }
    return 0;
}
```

```
$>gcc -Wall -std=c99 -DOPT= z.c
$>time -p a.out
real 23.81
user 23.66
sys 0.02

$>gcc -Wall -std=c99 -DOPT=register
$>time -p a.out
real 12.95
user 12.88
sys 0.01
```

MAKEFILE

MAKE

▷ L'utilitaire `make` d'Unix permet d'automatiser la compilation d'un programme fragmenté en plusieurs fichiers sources

▷ L'idée principale de `make` est d'effectuer uniquement les étapes de compilation nécessaires à la création d'un exécutable

  ▷ Seuls les fichiers modifiés sont recompilés

▷ La commande `make` recherche par défaut dans le répertoire courant un fichier de nom `makefile`, ou `Makefile` si elle ne le trouve pas

▷ Ce fichier spécifie les dépendances entre les différents fichiers sources, objets et exécutables

LE FICHIER MAKEFILE

▷ Un fichier `Makefile` est composé d'une liste de règles de dépendance de la forme :

```
cible: liste de dependances
<tabulation> commandes UNIX
```

▷ La première ligne spécifie un fichier cible, puis la liste des fichiers dont il dépend (séparés par des espaces)

▷ Les lignes suivantes, qui commencent par une tabulation, indiquent les commandes Unix à exécuter dans le cas où l'un des fichiers de dépendance est plus récent que le fichier cible

EXEMPLE DE MAKEFILE

```
## Premier exemple de Makefile
ex1: main.c sum.c sum.h
    gcc -std=c99 -Wall -o ex1 main.c sum.c
```

▷ L'exécutable `ex1` dépend des deux fichiers sources `sum.c` et `main.c`, ainsi que du fichier en-tête `sum.h`

▷ Il résulte de la compilation de ces deux fichiers en suivant la norme c99 avec tous les avertissements possibles

▷ Les commentaires sont précédés du caractère ♯

EXEMPLE DE MAKEFILE

```
## Premier exemple de Makefile
ex1: main.c sum.c sum.h
    gcc -std=c99 -Wall -o ex1 main.c sum.c
```

▷ Pour effectuer la compilation et obtenir un fichier cible, on lance la commande **make** suivie du nom du fichier cible souhaité

- ▷ `$> make ex1`

EXEMPLE DE MAKEFILE

```
## Premier exemple de Makefile
ex1: main.c sum.c sum.h
    gcc -std=c99 -Wall -o ex1 main.c sum.c
```

▷ Par défaut, si aucun fichier cible n'est spécifié au lancement de `make`, c'est la première cible du fichier `Makefile` qui est prise en compte

▷ Au premier lancement de la commande `make`, la compilation est effectuée puisque le fichier exécutable `ex1` n'existe pas

```
% make
gcc -std=c99 -Wall -o ex1 main.c sum.c
```

EXEMPLE DE MAKEFILE

```
## Premier exemple de Makefile
ex1: main.c sum.c sum.h
    gcc -std=c99 -Wall -o ex1 main.c sum.c
```

▷ Par défaut, si aucun fichier cible n'est spécifié au lancement de `make`, c'est la première cible du fichier `Makefile` qui est prise en compte

▷ Sans modification des fichiers sources ou en-tête, lors d'un nouveau lancement de la commande `make`, la compilation n'est pas effectuée puisque le fichier `ex1` est plus récent que les trois fichiers dont il dépend

```
% make
make: 'ex1' is up to date.
```

A BETTER MAKEFILE

▷ Afin de profiter pleinement des possibilités offertes par la compilation séparée, il est préférable de diviser la compilation d'un projet en sous-étapes

```
## Better example of Makefile
ex1: sum.o main.o
    gcc -o ex1 main.o sum.o
main.o : main.c sum.h
    gcc -std=c99 -Wall -c main.c
sum.o : sum.c sum.h
    gcc -std=c99 -Wall -c sum.c
```

▷ Les fichiers objet `main.o` et `sum.o` dépendent respectivement des fichiers sources `main.c` et `sum.c`, et du fichier entête `sum.h`

A BETTER MAKEFILE

▷ Afin de profiter pleinement des possibilités offertes par la compilation séparée, il est préférable de diviser la compilation d'un projet en sous-étapes

```
## Better example of Makefile
ex1: sum.o main.o
    gcc -o ex1 main.o sum.o
main.o : main.c sum.h
    gcc -std=c99 -Wall -c main.c
sum.o : sum.c sum.h
    gcc -std=c99 -Wall -c sum.c
```

▷ Les fichiers objet sont obtenus en effectuant la compilation de ces fichiers sources sans édition de liens

A BETTER MAKEFILE

▷ Afin de profiter pleinement des possibilités offertes par la compilation séparée, il est préférable de diviser la compilation d'un projet en sous-étapes

```
## Better example of Makefile
ex1: sum.o main.o
    gcc -o ex1 main.o sum.o
main.o : main.c sum.h
    gcc -std=c99 -Wall -c main.c
sum.o : sum.c sum.h
    gcc -std=c99 -Wall -c sum.c
```

▷ Le fichier exécutable `ex1` est obtenu en effectuant l'édition de liens des fichiers `sum.o` et `main.o`

A BETTER MAKEFILE

```
## Better example of Makefile
ex1: sum.o main.o
    gcc -o ex1 main.o sum.o
main.o : main.c sum.h
    gcc -std=c99 -Wall -c main.c
sum.o : sum.c sum.h
    gcc -std=c99 -Wall -c sum.c
```

▷ Au premier lancement de la commande **make**, la compilation est effectuée puisque le fichier exécutable **ex1** n'existe pas

```
% make
gcc -std=c99 -Wall -c sum.c
gcc -std=c99 -Wall -c main.c
gcc -o ex1 main.o sum.o
```

A BETTER MAKEFILE

```
## Better example of Makefile
ex1: sum.o main.o
    gcc -o ex1 main.o sum.o
main.o : main.c sum.h
    gcc -std=c99 -Wall -c main.c
sum.o : sum.c sum.h
    gcc -std=c99 -Wall -c sum.c
```

▷ Si l'on modifie le fichier `sum.c`, le fichier `main.o` est encore à jour donc seules deux des trois commandes sont exécutées

```
% make
gcc -std=c99 -Wall -c sum.c
gcc -o ex1 main.o sum.o
```

▷ L'option `-MM` de gcc permet de déterminer facilement les dépendances entre fichiers

```
% gcc -MM main.c
main.o: main.c sum.h
```

MR PROPRE

▷ On rajoute habituellement dans un fichier **`Makefile`** une cible appelée **`clean`** permettant de supprimer les fichiers intermédiaires produits par la compilation

```
## Better example of Makefile
ex1: sum.o main.o
    gcc -o ex1 main.o sum.o
main.o : main.c sum.h
    gcc -std=c99 -Wall -c main.c
sum.o : sum.c sum.h
    gcc -std=c99 -Wall -c sum.c
clean:
    rm -f ex1 *.o
```

MACROS ET ABBRÉVIATIONS

▷ On peut utiliser un certain nombre de macros de la forme `nom_de_macro = valeur_de_la_macro`

▷ Lors de l'appel à la commande `make` toutes les occurences du type `$(nom de macro)` dans le `Makefile` sont remplacées par la valeur de la macro

```
## An even better Makefile
# Compiler definition
CC = gcc
# Flags to be used for .o generation
CFLAGS = -Wall -std=c99 -c
ex1: sum.o main.o
    $(CC) -o ex1 main.o sum.o
main.o : main.c sum.h
    $(CC) $(CFLAGS) main.c
sum.o : sum.c sum.h
    $(CC) $(CFLAGS) sum.c
clean:
    rm -f ex1 *.o
```

QUELQUES OPTIONS DE GCC

OPTIONS DE COMPILATION

▷ Le compilateur `gcc` se lance par la commande

`gcc [options] fichier.c [-llibrairies]`

▷ Par défaut, le fichier exécutable s'appelle `a.out`

▷ Les éventuelles librairies sont déclarées via l'option `-llibrairie`

▷ Le fichier `liblibrairie.a` est recherché dans le répertoire `/usr/lib/`

▷ Par exemple, les fonctions mathématiques de la librairie `math.h` ne sont accessibles qu'après inclusion de la librairie `libm.a` (qui nécessite donc d'ajouter l'option `-lm`)

OPTIONS DE COMPILATION

▷ Le compilateur `gcc` se lance par la commande

```
gcc [options] fichier.c [-llibrairies]
```

▷ `-c` permet de produire un fichier objet sans procéder à l'édition de liens

▷ `-E` n'active que le préprocesseur et affiche le résultat

▷ `-g` produit un fichier objet contenant des informations symboliques nécessaires au débogueur

OPTIONS DE COMPILATION

▷ Le compilateur `gcc` se lance par la commande

```
gcc [options] fichier.c [-llibrairies]
```

▷ `-Inom-de-repertoire` ajoute un répertoire où rechercher des fichiers d'en-tête

▷ `-Lnom-de-repertoire` ajoute un répertoire où rechercher des librairies pré-compilés

▷ `-o nom-de-fichier` spécifie le nom du fichier produit

OPTIONS DE COMPILATION

▷ Le compilateur `gcc` se lance par la commande

```
gcc [options] fichier.c [-llibrairies]
```

▷ `-O, -O1, -O2, -O3` autorise le compilateur à optimiser le code produit suivant un niveau d'optimisation (`-O1` : faible, `-O3` : élevé)

▷ `-S` n'active que le préprocesseur et le compilateur et produit un fichier assembleur

▷ `-Wall` fournit tous les messages d'avertissement

▷ `-Werror` fournit tous les messages d'avertissement comme des erreurs de compilation

DOGGY BAG

TO TAKE AWAY ...

▷ Des constantes symboliques doivent être définies

▷ Le code doit être factorisé et fragmenté en plusieurs fichiers

▷ Les fichiers d'en-tête doivent être protégés par des macros

▷ La visibilité d'une variable ou d'une fonction peut être contrôlée à l'aide des mot clefs `extern` et `static`

▷ Tous les fichiers résultant d'une étape de la compilation d'un programme sont accessibles via des options de `gcc`

Questions?