Compilation séparée et Makefile

Programmation séquentielle en C, 2021-2022

Orestis Malaspinas (A401), ISC, HEPIA

2021-11-02

Inspirés des slides de F. Glück

Prototypes de fonctions (1/3)

```c
// Prototype, pas d'implémentation, juste la doc
int sum(int size, int tab[size]);
```

Principes généraux de programmation

- Beaucoup de fonctionnalités dans un code $\Rightarrow$ Modularisation.
- Modularisation du code $\Rightarrow$ écriture de fonctions.
- Beaucoup de fonctions $\Rightarrow$ regrouper les fonctions dans des fichiers séparés.

Mais pourquoi?

- Lisibilité.
- Raisonnement sur le code.
- Débogage.

Exemple

- Libraire `stdio.h`: `printf()`, `scanf()`, ...

Prototypes de fonctions (2/3)

- Prototypes de fonctions nécessaires quand:
  1. Utilisation de fonctions dans des fichiers séparés.
  2. Utilisation de librairies.
- Un prototype indique au compilateur la signature d'une fonction.
- On met les prototypes des fonctions **publiques** dans des fichiers *headers*, extension `.h`.
- Les *implémentations* des fonctions vont dans des fichier `.c`.

Prototypes de fonctions (3/3)

Fichier header

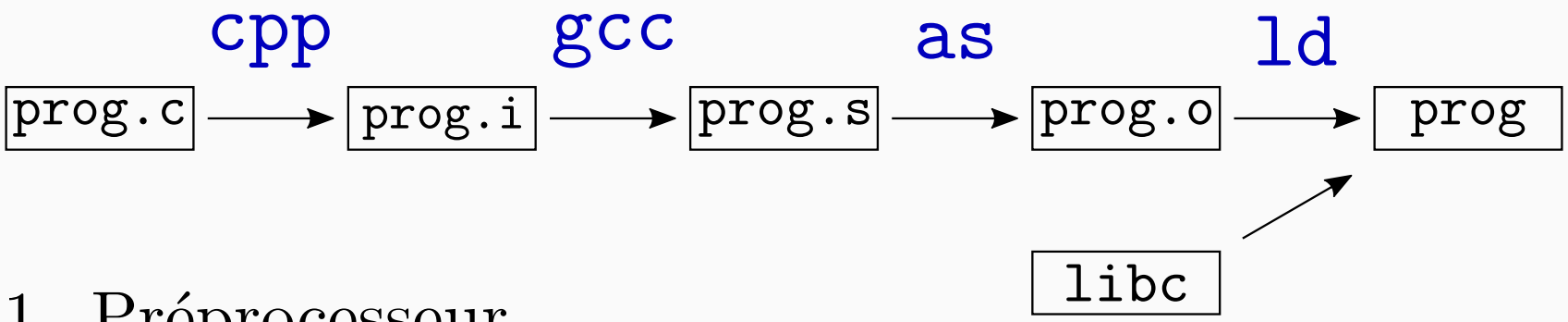
- Porte l'extension `.h`
- Contient:
  - définitions des types
  - prototypes de fonctions
  - macros
  - directives préprocesseur (cf. plus loin)
- Utilisé pour décrire **l'interface** d'une librairie ou d'un module.
- Un fichier `C` (extension `.c`) utilise un header en *l'important* avec la directive *`#include`*:

```
#include <stdio.h> // libraire dans LD_LIBRARY_PATH
#include "chemin/du/prototypes.h" // chemin explicite
```

Génération d'un exécutable (1/5)

Un seul fichier source

```
gcc -o prog prog.c
```



1. Préprocesseur
2. Compilation assembleur
3. Compilation code objet
4. Édition des liens

Figure 1: Étapes de génération.

Génération d'un exécutable (2/5)

```
gcc proc.c -o prog
```

1. **Précompilation:** `gcc` appelle `cpp`, le préprocesseur qui effectue de la substitution de texte (`#define`, `#include`, macros, ...) et génère le code `C` à compiler, portant l'extension `.i` (`prog.i`).
2. **Compilation assembleur:** `gcc` compile le code C en code assembleur, portant l'extension `.s` (`prog.s`).
3. **Compilation code objet:** `gcc` appelle `as`, l'assembleur, qui compile le code assembleur en code machine (code objet) portant l'extension `.o` (`prog.o`).
4. **Édition des liens:** `gcc` appelle `ld`, l'éditeur de liens, qui lie le code objet avec les librairies et d'autres codes objet pour produire l'exécutable final (`prog`).

Les différents codes intermédiaires sont effacés.

Génération d'un exécutable (3/5)

Plusieurs fichiers sources

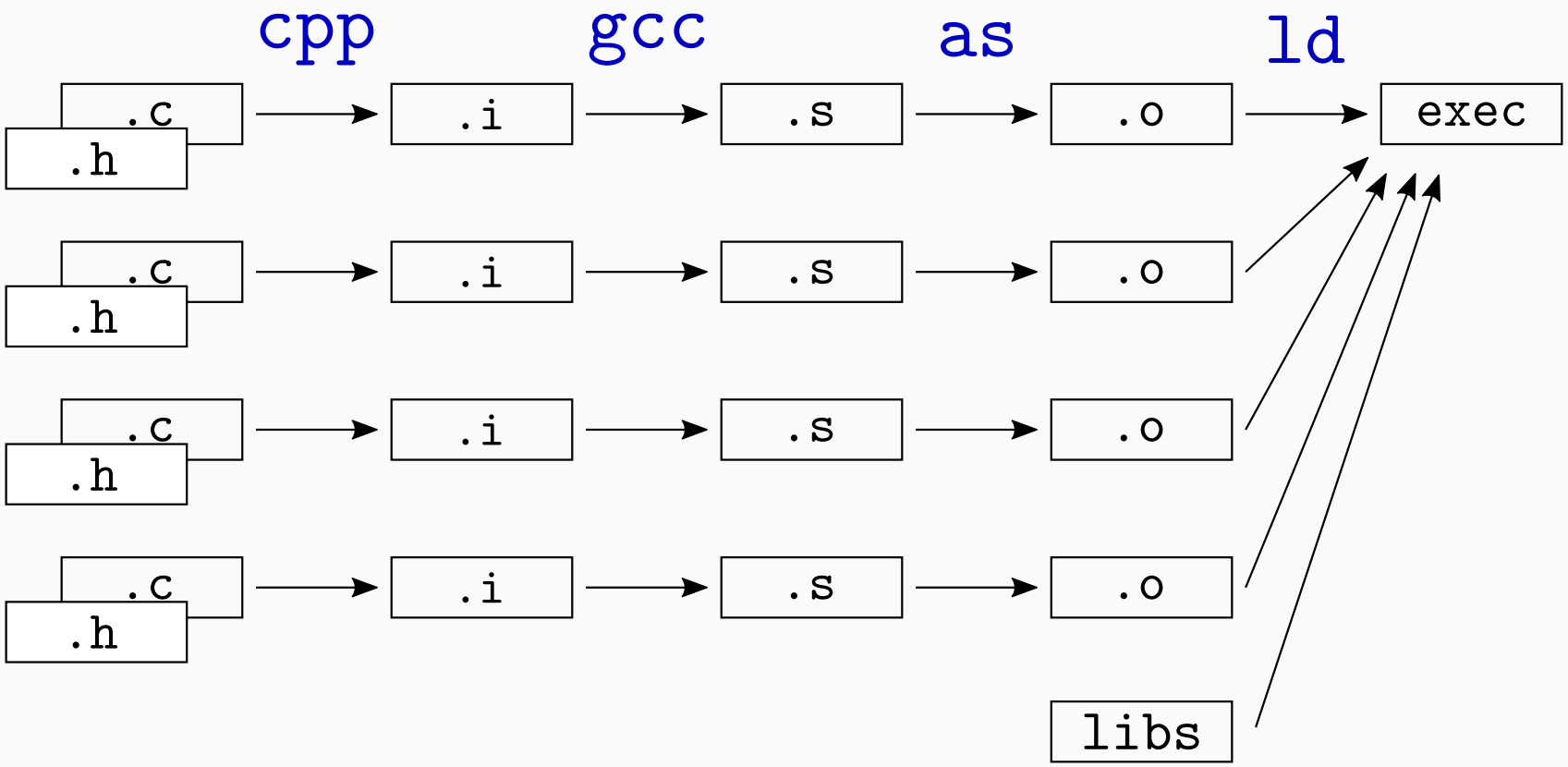


Figure 2: Étapes de génération, plusieurs fichiers.

Génération d'un exécutable (4/5)

main.c

```c
#include <stdio.h>
#include "sum.h"
int main() {
  int tab[] = {1, 2, 3, 4};
  printf("sum: %d\n", sum(tab, 4));
  return 0;
}
```

sum.h

```c
#ifndef _SUM_H_
#define _SUM_H_
int sum(int tab[], int n);
#endif
```

sum.c

```c
#include "sum.h"
int sum(int tab[], int n) {
  int s = 0;
  for (int i = 0; i < n; i++) {
    s += tab[i];
  }
  return s;
}
```

Génération d'un exécutable (5/5)

La compilation séparée se fait en plusieurs étapes.

Compilation séparée

1. Générer séparément les fichiers `.o` avec l'option `-c`.
2. Éditer les liens avec l'option `-o` pour générer l'exécutable.

Exemple

- Création des fichiers objets, `main.o` et `sum.o`

```
$ gcc -Wall -Wextra -std=c11 -c main.c
$ gcc -Wall -Wextra -std=c11 -c sum.c
```

- Édition des liens

```
$ gcc main.o sum.o -o prog
```

Préprocesseur (1/2)

Généralités

- Première étape de la chaîne de compilation.
- Géré automatiquement par `gcc` ou `clang`.
- Lit et interprète certaines directives:
  1. Les commentaires (`//` et `/* ... */`).
  2. Les commandes commençant par **#**.
- Le préprocesseur ne compile rien, mais subtitue uniquement du texte.

La directive `define`

- Permet de définir un symbole:

```
#define PI 3.14159
#define _SUM_H_
```

- Permet de définir une macro.

```
#define NOM_MACRO(arg1, arg2, ...) [code]
```

Préprocesseur (2/2)

La directive `include`

- Permet d'inclure un fichier.
- Le contenu du fichier est ajouté à l'endroit du *#include*.
- Inclusion de fichiers "globaux" ou "locaux"

```
#include <file.h>       // LD_LIBRARY_PATH
#include "other_file.h" // local path
```

- Inclusions multiples peuvent poser problème: définitions multiples. Les headers commencent par:

```
#ifndef _VAR_
#define _VAR_
/* commentaires */
#endif
```

Introduction à `make`

A quoi ça sert?

- Automatiser le processus de conversion d'un type de fichier à un autre, en *gérant les dépendances*.
- Effectue la conversion des fichiers qui ont changé uniquement.
- Utilisé pour la compilation:
  - Création du code objet à partir des sources.
  - Création de l'exécutable à partir du code objet.
- Tout "gros" projet utilise `make` (pas uniquement en `C`).

Utilisation de `make`

Le programme `make` exécutera la série d'instruction se trouvant dans un `Makefile` (ou `makefile` ou `GNUmakefile`).

Le `Makefile`

- Contient une liste de *règles* et *dépendances*.
- Règles et dépendances construisent des *cibles*.
- Ici utilisé pour compiler une série de fichiers sources

```
$ gcc -c example.c # + plein d'options..
$ gcc -o example exemple.o # + plein d'options
```

`Makefile`

```
example: example.o
    gcc -o example example.o

exmaple.o: exmaple.c example.h
    gcc -c example.c
```

Terminal

```
$ make
gcc -c example.c
gcc -o example example.o
```

Syntaxe d'un `Makefile` (1/4)

```
example: example.c example.h
	gcc example.c -o example
```

Figure 3: Un exemple simple de `Makefile`.

Syntaxe d'un `Makefile` (2/4)

cible (target)

```
example: example.c example.h
	gcc example.c -o example
```

Figure 4: La cible.

Syntaxe d'un `Makefile` (3/4)

dépendances (dependencies)

```
example: example.c example.h
	gcc example.c -o example
```

Figure 5: Les dépendances.

Syntaxe d'un `Makefile` (4/4)

```
example: example.c example.h
	gcc example.c -o example
```

règle (rule)

Figure 6: La règle.

Principe de fonctionnement

1. `make` cherche le fichier `Makefile`, `makefile` ou `GNUmakefile` dans le répertoire courant.
2. Par défaut exécute la première cible, ou celle donnée en argument.
3. Décide si une cible doit être régénérée en comparant la date de modification (on recompile que ce qui a été modifié).
4. Regarde si les dépendances doivent être régénérées:
   - Oui: prend la première dépendance comme cible et recommence à 3.
   - Non: exécute la règle.

`make` a un comportement **récursif**.

Exemple avancé

Makefile

```
hello: hello.o main.o
    gcc hello.o main.o -o hello

hello.o: hello.c hello.h
    gcc -Wall -Wextra -c hello.c

main.o: main.c
    gcc -Wall -Wextra -c main.c

clean:
    rm -f *.o hello

rebuild: clean hello
```

Un graph complexe

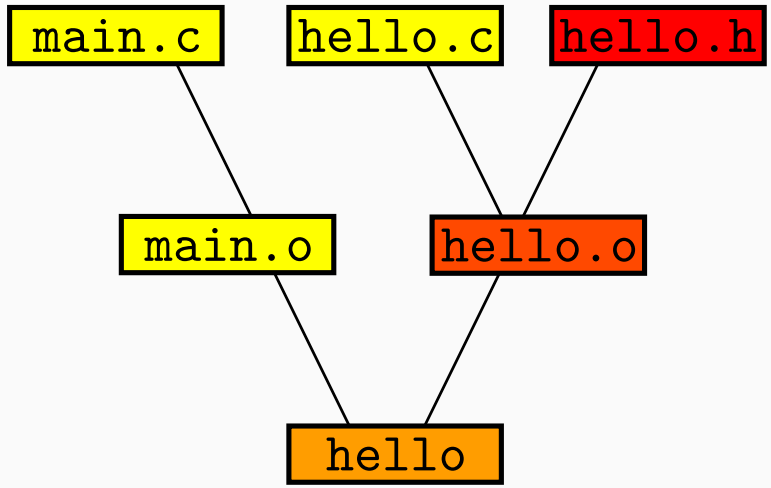


Figure 7: Makefile complexe.

Factorisation

Ancien `Makefile`

```
hello: hello.o main.o
    gcc hello.o main.o -o hello

hello.o: hello.c hello.h
    gcc -Wall -Wextra -c hello.c

main.o: main.c
    gcc -Wall -Wextra -c main.c

clean:
    rm -f *.o hello

rebuild: clean hello
```

Nouveau `Makefile`

```
CC=gcc -Wall -Wextra

hello: hello.o main.o
    $(CC) $^ -o $@

hello.o: hello.c hello.h
    $(CC) -c $<

main.o: main.c
    $(CC) -c $<

clean:
    rm -f *.o hello
```

Variables

Variables utilisateur

- Déclaration

```
id = valeur
id = valeur1 valeur2 valeur3
```

- Utilisation

```
$(id)
```

- Déclaration à la ligne de commande

```
make CFLAGS="-O3 -Wall"
```

Variables internes

- $@ : la cible
- $^ : la liste des dépendances
- $< : la première dépendance
- $* : le nom de la cible sans extension