

# Chapitre 3 : Compilation et automatisaton

## Construction et maintenance de logiciels

Guy Francoeur

basé sur du matériel pédagogique d'Alexandre Blondin Massé, professeur

**UQÀM** | **Département d'informatique**

# Table des matières

1. Compilation

2. Makefiles

# Table des matières

1. Compilation

2. Makefiles

# Réalisation d'un exécutable

1. **Édition** du code source à l'aide d'un **éditeur de texte** ou d'un **environnement de développement**. L'extension du fichier source est **.c**.
2. **Compilation** du code source, traduction du langage C en **langage machine**. Le compilateur indique les **erreurs de syntaxe**, mais ignore les **bibliothèques** déjà compilées appelées par le programme. Le compilateur génère du code machine en mémoire ou un fichier avec l'extension **.o**.
3. **Édition de liens**. Le code machine de différents fichiers **.o** est assemblé pour former un fichier **binaire**. Le résultat porte le nom de l'option -o.
4. **Exécution du programme**. Soit en **ligne de commande** ou en **double-cliquant** sur l'icône du fichier binaire.

# Étapes de compilation

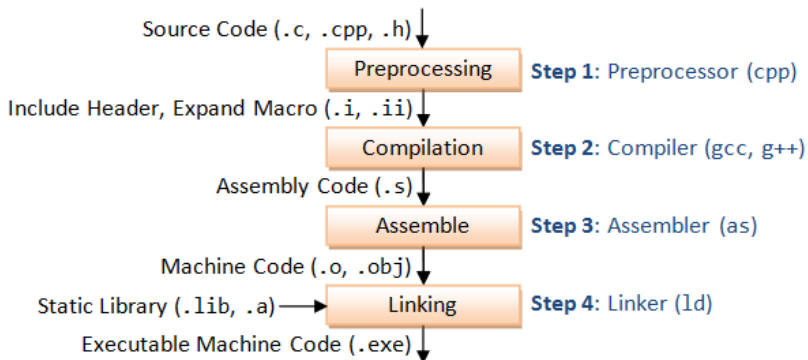


Figure: étapes de compilation

# Étapes de compilation

1. Préprocesseur : inclus les fichiers d'entête et remplace les MACROS;
2. Compilation : vérifier tout le code pour une syntaxe correcte et produire du code assembleur (.s);
3. Assembleur : Produire du code objet (.o binaire);
4. Édition de liens : Lier tout le code objet et les librairies statiques .a nécessaire et produire l'exécutable;

# Exemple de programme C

```
1 // Directives au preprocesseur
2 #include <stdio.h>
3 #define DEBUT -2
4 #define FIN 10
5 #define MSG "Programme de demonstration\n"
6
7 // Prototypes de fonctions
8 int carre(int x);
9 int cube(int x);
10
11 // Fonction main
12 int main() {
13     int i;
14     printf(MSG);
15     for (i = DEBUT; i <= FIN; i++) {
16         printf("%d carre: %d cube: %d\n", i, carre(i), cube(i));
17     }
18     return 0;
19 }
20
21 // Implementation
22 int cube(int x) {
23     return x * carre(x);
24 }
25
26 int carre(int x) {
27     return x * x;
28 }
```

## Exemple de compilation (1/2)

- ▶ Reprenons le fichier **exemple.c**
- ▶ On peut le **compiler** en exécutable en un appel :

```
$ gcc exemple.c
```

ce qui produit le fichier **a.out**.

- ▶ Puis ensuite, on l'**exécute** :

```
$ ./a.out
```

Programme de démonstration

-2 carré: 4 cube: -8

-1 carré: 1 cube: -1

0 carré: 0 cube: 0

1 carré: 1 cube: 1

2 carré: 4 cube: 8



## Exemple de compilation (2/2)

- ▶ Revenons à la compilation en un appel de GCC afin de produire un exécutable nommé :

```
$ gcc -o exemple exemple.c
```

- ▶ Toutes les étapes (4) seront exécuté en mémoire et seulement le fichier exécutable sera produit.

# Compilation avec les .o

- ▶ Il est possible de compiler en **deux appels**;
- ▶ D'abord, de **.c** vers **.o** :

```
$ gcc -c exemple.c
```

ce qui produit le fichier **compilé** (objet) **exemple.o**.

- ▶ Puis ensuite, la commande

```
$ gcc -o exemple exemple.o
```

produit un fichier **exécutable** nommé **exemple**.

- ▶ Il s'exécute simplement en entrant

```
$ ./exemple
```

- ▶ On a vu un peu les **deux appels** pour créer un **exécutable en C** :
  - ▶ Pourquoi compiler en une étape (un appel) ?
  - ▶ Pourquoi compiler en utilisant deux étapes (plusieurs appels) ?
- ▶ Pour répondre à ces questions, nous allons voir les Makefile et tenter une réponse par la suite.
- ▶ On met donc les questions sur la glace!

# Table des matières

1. Compilation

2. Makefiles

- ▶ Existent depuis la fin des **années '70**.
- ▶ Gèrent les **dépendances** entre les différentes composantes d'un programme;
- ▶ Automatisent la **compilation** en **minimisant** le nombre d'étapes;
- ▶ Malgré qu'ils soient **archaïques**, ils sont encore **très utilisés** (et le seront sans doute pour **très longtemps** encore);
- ▶ Certaines **limitations** des Makefiles sont corrigées par des outils comme **Autoconf** et **CMake**;
- ▶ Sa simplicité est probablement sa plus grande force.

# Exemple


- ▶ Un **Makefile** pour automatiser la compilation d'**exemple.c** :

```
1  exemple: exemple.o
2      gcc -o exemple exemple.o
3
4  exemple.o: exemple.c
5      gcc -c exemple.c
```

- ▶ La **syntaxe** est **stricte** et sous la forme :

<cible>: <dépendances>

<tab><commande>

(le caractère <tab>  est très important !)

# Exécution d'un Makefile

- Pour générer une cible contenue dans le **Makefile**, taper :

```
$ make exemple
```

- On obtient alors (dans le terminal) :

```
gcc -c exemple.c
```

```
gcc -o exemple exemple.o
```

et le fichier **.o** et l'**exécutable** sont produits.

Simplement pour compiler en un appel :

```
1  tp1d: tp1.c
2      gcc -g -o tp1d tp1.c
```

Pourquoi le -g ?

Plus complet, toujours en une étape (un appel de GCC) :

```
1  tp1: tp1.c
2      gcc -std=c11 -Wall -Wextra -o tp1 tp1.c
```



- ▶ Il est possible d'utiliser des **variables** dans un Makefile;
- ▶ Une variable est **initialisée** en écrivant :

```
1 <nom variable> = <valeur>
```

- ▶ On récupère ensuite son **contenu** de cette façon :

```
1 $(<nom variable>)
```

# Exemple

```
1 #Makefile_01 version de base
2 CC = gcc
3 CFLAGS = -Wall
4 SRC = tp1.c
5 OBJ = tp1.o
6 EXEC = tp1
7
8 $(EXEC): $(OBJ)
9     $(CC) $(CFLAGS) -o $(EXEC) $(OBJ)
10
11 $(OBJ): $(SRC)
12     $(CC) $(CFLAGS) -c $(SRC)
```

```
1 #Makefile_02 version simplifi
2 FILENAME = tp1
3 CFLAGS = -Wall -std=c11
4
5 $(FILENAME): $(FILENAME).o
6     gcc $(CFLAGS) -o $(FILENAME) $(FILENAME).o
7
8 $(FILENAME).o: $(FILENAME).c
9     gcc $(CFLAGS) -c $(FILENAME).c
```

- ▶ Il est également possible de définir des **cibles** qui ne sont pas des noms de fichier;
- ▶ Quelques cibles classiques :
  - ▶ **clean** : pour nettoyer le répertoire du projet;
  - ▶ **all** : pour générer l'ensemble du projet;
  - ▶ **install** : pour installer le projet sur la machine;
  - ▶ **test** : pour lancer une suite de tests;
  - ▶ **doc** : pour générer la documentation, etc.

# La cible .PHONY

- Lorsqu'on utilise des cibles qui ne génèrent pas de fichier sur disque, il est requis de les ajouter dans une cible **.PHONY**: comme ceci :

```
1 #Makefile_03 version avanc
2 FILENAME = tpl
3 CFLAGS = -Wall -std=c99 -pedantic -Werror=vla
4
5 $(FILENAME): $(FILENAME).o
6     gcc $(CFLAGS) -o $(FILENAME) $(FILENAME).o
7
8 $(FILENAME).o: $(FILENAME).c
9     gcc $(CFLAGS) -c $(FILENAME).c
10
11 .PHONY: clean
12
13 clean:
14     rm -f $(EXEC) $(FILENAME).o
```

# Nos questions sur la glace

- ▶ Est-ce vraiment nécessaire de compiler tous les fichiers de zéro chaque fois?
- ▶ Rappel : le processus de compilation comporte 4 étapes.
- ▶ Est-ce possible de reprendre après l'étape 3 pour certains fichiers ?
- ▶ Est-ce que le Makefile peut nous rendre la vie facile pour couper court ?
- ▶ Quels sont les avantages liés Makefile ?