Chapitre 2 — Le compilateur

Objectifs pédagogiques

À la fin de ce chapitre, l'étudiant devra être capable de :

- Comprendre le rôle du **compilateur** dans le développement C++.
- Utiliser le compilateur **GCC/G++** pour compiler des programmes.
- Comprendre et mettre en œuvre un **compilateur croisé** (x86/ARM).
- Différencier le **linkage statique et dynamique**.
- Déboguer un programme C++ avec des outils comme **gdb**.
- Créer et utiliser un **Makefile** pour automatiser la compilation.

1. Introduction au compilateur

1.1.Qu'est-ce qu'un compilateur ?

- C'est un programme qui transforme le **code source (C++)** en **code machine exécutable**.
- Étapes classiques :
    1. **Préprocessing** : traitement des directives #include, #define.
    2. **Compilation** : traduction du code C++ en **code assembleur**.
    3. **Assemblage** : assemble le code assembleur en **fichier objet (.o)**.
    4. **Linking (édition des liens)** : création de l'exécutable final, en liant les bibliothèques.

Exemple : compilation simple

```
g++ main.cpp -o mon_programme
```

- main.cpp → code source
- -o mon_programme → nom de l'exécutable

2. Compilateur open source : GCC / G++

2.1. GCC (GNU Compiler Collection)

- Compiler libre et multi-langages : C, C++, Fortran, etc.
- **G++** est le compilateur spécifique pour le C++.

Avantages

- Open source, disponible sur Linux, Windows (via MinGW), macOS.
- Optimisations performantes (-O1, -O2, -O3, -Ofast).
- Supporte toutes les normes modernes du C++ (C++98 à C++20).

2.2. Commandes courantes

Commande	Description
g++ main.cpp -o prog	Compile un programme simple
g++ -Wall -Wextra main.cpp -o prog	Active tous les warnings
g++ -std=c++17 main.cpp -o prog	Utilise la norme C++17
g++ -O2 main.cpp -o prog	Optimisation du code

3. Compilateur croisé (x86/ARM)

Un **compilateur croisé** produit du code exécutable pour une **architecture différente** de celle de la machine hôte.

- Exemple : compiler sur un PC x86 pour une carte ARM (Raspberry Pi).

Exemple : GCC croisé pour ARM

arm-linux-gnueabihf-g++ main.cpp -o prog_arm

- arm-linux-gnueabihf-g++ → compilateur croisé
- Permet de générer un exécutable ARM depuis un PC x86

Avantages

- Développement embarqué.
- Simulation et test sur hôte avant déploiement sur matériel cible.

4. Linkage dynamique et statique

4.1. Linkage statique

- Toutes les bibliothèques sont intégrées dans l'exécutable.
- Avantages : **exécutable autonome**, pas de dépendance externe.
- Inconvénients : taille plus grande.

g++ main.cpp libmath.a -o prog_static

- .a = bibliothèque statique

4.2. Linkage dynamique

- L'exécutable utilise des **bibliothèques partagées** au moment de l'exécution.
- Avantages : économie de mémoire, mise à jour des bibliothèques possible.
- Inconvénients : dépendances externes.

g++ main.cpp -L/usr/lib -lmylib -o prog_dynamic

- .so (Linux) ou .dll (Windows)

5. Débogage

5.1. Outils : gdb

- Permet d'exécuter le programme **pas à pas**, examiner les variables, suivre les bugs.

5.2. Commandes de base

g++ -g main.cpp -o prog # -g pour ajouter les symboles de debug

gdb prog # lance gdb

5.3. Dans gdb

Commande	Fonction
break main	poser un point d'arrêt à main
run	exécuter le programme
next	ligne suivante (pas entrer dans les fonctions)
step	ligne suivante (entre dans les fonctions)
print x	afficher la valeur de la variable x
continue	reprendre l'exécution jusqu'au prochain point d'arrêt

6. Makefile

- Fichier texte qui automatise la compilation.
- Évite de taper manuellement toutes les commandes gcc/g++.
- Exemple de Makefile simple :

```
CC = g++
CFLAGS = -Wall -std=c++17
SRC = main.cpp utils.cpp
OBJ = $(SRC:.cpp=.o)
TARGET = mon_programme

all: $(TARGET)

$(TARGET): $(OBJ)
    $(CC) $(CFLAGS) -o $(TARGET) $(OBJ)

%.o: %.cpp
    $(CC) $(CFLAGS) -c $< -o $@

clean:
    rm -f $(OBJ) $(TARGET)
```

Explications :

- all → cible par défaut
- $(OBJ) → fichiers objets générés
- $< → nom du fichier source
- clean → supprime exécutables et objets