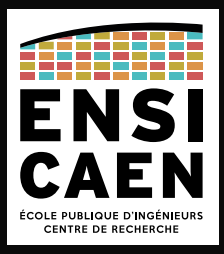


# Outils de développement logiciel

TP n° 5 : chaîne de compilation

Alain Lebret

2023-2024

**Objectif**

Appréhender la chaîne de compilation et ses différentes étapes à l'aide de l'outil ***GNU gcc***.
**Pré-requis** : chapitre 9
**Durée estimée** : 1 séance

Hormis pour l'édition de votre compte-rendu de séance, n'utilisez pas ***Vim*** jusqu'à l'exercice n° 6, et contentez-vous des commandes Unix rencontrées auparavant, ainsi que de nouvelles commandes qui pourraient vous être proposées.

## 1 Étapes de compilation

Vous avez vu dans le tout premier cours de langage C que pour "compiler" le programme défini dans le fichier "programme1.c" et générer l'exécutable `programme1`, il a fallu utiliser la commande `gcc` de la manière suivante :

```
$ gcc programme1.c -o programme1
```

En fait, la commande `gcc` est un **frontal** qui masque l'ensemble des étapes (et donc des commandes sous-jacentes) permettant, à partir d'un programme écrit en langage C, de générer un exécutable pour une architecture particulière et un système d'exploitation donné. Nous allons détailler ces différentes étapes tout au long de ce TP.

Dans la plupart des cas, il est préférable de passer directement par un appel à **`gcc`** qui prend en compte les spécificités des commandes sous-jacentes sur une machine et un système donné. En effet, un changement de système peut très bien amener à des changements de dossiers ou d'options lors de l'appel de ces commandes, ce qui risquera d'engendrer des modifications impactant le travail du développeur.

### Étape de précompilation

L'étape de précompilation est réalisée par le programme appelé **préprocesseur** (commande **cpp**). Lors de cette étape, le préprocesseur réalise des transformations purement textuelles (remplacement de chaînes de caractères, inclusion de fichiers sources, etc.). On peut soit appeler le préprocesseur directement, soit passer par le frontal `gcc` et son option **–E** :

```
$ gcc –E –o programme1.i programme1.c
$ cpp –o programme1.i programme1.c
```

### Exercice n° 1

1. Déplacez-vous dans votre dossier "odl" et créez le sous-dossier "tp05", puis placez-vous dans ce dernier. "tp05" sera votre dossier de travail pour cette séance et vous y effectuerez toutes vos actions.
2. Dans le dossier de travail, copiez le fichier "vector.c" que vous trouverez dans le dossier "ressources-odl-fisa/tp05/", puis visualiser son contenu.
3. En utilisant l'option –E de `gcc`, générez le code précompilé dans le fichier "vector.i". Visualisez le contenu du fichier "vector.i". Réitérez en appelant directement le préprocesseur afin de générer le code préprocessé dans un fichier nommé "vector-bis.i", puis vérifiez que les fichiers produits sont bien identiques dans les deux cas (commande **`diff`** par exemple).
4. Comparez le nombre de lignes de "vector.c" et de "vector.i" (commande **`wc`**).
5. La macro `N` est-elle toujours présente dans "vector.i" (au choix les commandes **`grep`**, **`tail`**, etc.) ?
6. Les commentaires sont-ils toujours là ? (***idem***)
7. Les directives `#include <xxx.h>` sont-elles toujours présentent ? Par quoi ont-elles été remplacées ? (***idem***)
8. Visualisez le contenu du fichier "vector.i" à l'aide de la commande **`less`**.

### Étape de traduction (ou compilation)

Dans cette étape, le fichier engendré par le préprocesseur est traduit en assembleur (après vérification de sa syntaxe), c.-à-d. en une suite d'instructions associées aux fonctionnalités du microprocesseur (faire une addition, etc.).

On peut appeler soit le compilateur **`cc1`** directement, soit passer par le frontal `gcc` et son option **–S** :

```
$ /usr/lib/gcc/x86_64-linux-gnu/7/cc1 -o programme1.s programme1.i
$ gcc -S -o programme1.s programme1.i
```

Comme vous pouvez le constater ci-dessus, l'utilisation du compilateur `cc1` nécessite d'indiquer son chemin absolu, et ce dernier peut évoluer au gré des changements de version de l'outil de compilation ou encore du système.

### Exercice n° 2

1. Utilisez l'option `-S` de `gcc` afin de générer le fichier "vector.s" (en langage assembleur) à partir du fichier "vector.i" issu en sortie du préprocesseur.
2. Visualisez le type du fichier obtenu à l'aide de la commande **`file`** qui est intéressante pour obtenir des informations sur le type du fichier.
3. Visualisez le contenu du fichier "vector.s" obtenu.
4. Vérifiez que l'appel direct du compilateur **`cc1`** produit le même code assembleur (commande **`diff`**). Si ce n'est pas le cas, qu'a-t-il bien pu se passer ?

### Étape d'assemblage

Un programme écrit en langage assembleur est constitué d'une suite d'instructions spécifiques à une architecture donnée (microprocesseur). Ces instructions sont stockées dans le segment de texte (commence par la directive ".text") et exécutées séquentiellement. Le point d'entrée du programme correspond à l'étiquette marquant la première instruction de ce segment.

Pour les curieux : le site https://gcc.godbolt.org permet de visualiser le code assembleur obtenu à partir d'un code en langage C, ou C++, etc., et ce pour différentes architectures et différents compilateurs.

### Exercice n° 3

1. Utilisez la commande **`as`** (assembleur) afin de produire le fichier objet "vector.o" à partir du fichier "vector.s" précédent.

2. Mettez en oeuvre la commande `objdump` et l'option `–d` afin de désassembler le fichier "vector.o". Le code objet est affiché à gauche, et le code désassemblé à droite. Comparez avec le code assembleur du départ.

### Étape d'édition des liens

L'étape d'édition des liens permet de grouper le ou les fichiers objets obtenus lors des étapes précédentes avec les codes des différentes fonctions utilisées (par exemple le code de la fonction `printf()`, ces codes étant, pour les fonctions couramment utilisées, contenus dans les différentes bibliothèques fournies avec le système (par exemple `libc.a` ou `libc.so` pour les fonctions relevant de la bibliothèque standard du C). L'édition des liens est réalisée par l'outil **`ld`** et génère un fichier exécutable.

Par exemple, l'édition de liens permettant de produire l'exécutable `programme1` à partir du fichier objet "programme1.o" sera réalisée par une des deux commandes ci-dessous (sur d'autres systèmes que ceux de l'ENSICAEN, celle utilisant `ld` sera sans aucun doute à adapter) :

```
$ ld –static –o programme1 –L`gcc –print–file–name=`
 ↪  /usr/lib/x86_64–linux–gnu/crt1.o /usr/lib/x86_64–linux–gnu/crti.o programme1.o
 ↪  /usr/lib/x86_64–linux–gnu/crtn.o ––start–group –lc –lgcc –lgcc_eh ––end–group
$ gcc –o programme1 programme1.o
```

### Exercice n° 4

1. Générez le programme exécutable `vector` à partir du fichier objet "vector.o" en utilisant les deux méthodes.
2. Exécutez le programme `vector`. D'après le résultat affiché, que pourrait-on dire des vecteurs **a** et **b** ? Est-ce bien le cas ? Sinon passez à la suite.

### Options de l'étape de traduction

Nous avons pu remarquer en lançant le programme `vector` que celui-ci ne fournissait pas le résultat escompté. Lors de la phase de traduction, nous avons omis un ensemble d'options qui devraient être obligatoirement ajoutées
de manière à limiter ce type de déconvenues : les options permettant l'affichage des avertissements (**`–Wall`** et **`–Wextra`**), ainsi que celles d'une vérification de la norme ANSI (**`–ansi`**) et d'une analyse très scrupuleuse (**`–pedantic`**).

### Exercice n° 5

1. Sans repasser par toutes les étapes précédentes, relancez la compilation de “vector.c” en ajoutant les options manquantes.
2. Corrigez les anomalies, recompilez et vérifiez. Vous justifierez dans votre compte-rendu la correction des anomalies en vous servant de la trace d’appel à ***gcc*** que vous inclurez (trace après options et avant correction évidemment).

### Étape d’optimisation

Nous rappelons les différentes niveaux d’optimisation :

| Option | Description |
|---|---|
| `-O0` | Aucune optimisation (utilisée lorsque phase de déverminage). |
| `-O` ou `-O1` | Compromis temps / espace. |
| `-O2` | Optimisation en vitesse, en se permettant de prendre plus de place de mémoire. Par exemple, dupliquer le contenu des itérations en autant de lignes. |
| `-O3` | Encore plus de vitesse, en utilisant des astuces d’optimisation. Attention, cela peut changer le code (par exemple, changer l’ordre de lignes indépendantes pour rapprocher des lignes utilisant une même variable afin de la garder dans le registre). |
| `-Os` | Optimise la taille – correspond à `-O2` sans augmentation de taille. |
| `-Og` | Nouveau niveau d’optimisation général compatible avec le déverminage. |
| `-Ofast` | Consiste en `-O3` et `-ffast-math`. |

### Exercice n° 6

1. Créez le fichier “sum.c” contenant le code suivant :

```
int sum(int n) {
  int i;
  int sum;

  sum = 0;
```

```
  for (i = 0; i <= n; i++) {
    sum += i;
  }

  return sum;
}
```

2. Créez les sous-dossiers "sum_output/O0" à "sum_output/O3" et "sum_output/Og" qui vous serviront à stocker les fichiers intermédiaires pour les différents niveaux d'optimisation (on laissera de côté les niveaux d'optimisation `Os` et `Ofast`).
3. Générez le code assembleur (option `-S`) avec le niveau d'optimisation `Og` :

   ```
   $ gcc -Og -S -o ./sum_output/Og/sum.s sum.c
   ```

   puis examinez le résultat.
4. Répétez l'opération en ajoutant l'option `-fdump-tree-all` qui permet de générer les fichiers intermédiaires et de compter le nombre de "passes" réalisées par le compilateur (nombre de fichiers intermédiaires générés).
5. Renouvelez les opérations en utilisant les niveaux d'optimisation `O0` [1], `O1`, `O2` et `O3`, puis examiner le code assembleur obtenu et relever le nombre de passes. Conclure.
6. Remplacez l'option `-S` par `-c` afin de produire du code objet ("sum.o") :

   ```
   $ gcc -Og -c sum.c
   ```

   puis utilisez la commande `objdump` afin de désassembler le code objet :

   ```
   $ objdump -d sum.o
   ```

   Le code objet est affiché à gauche, et le code désassemblé à droite. Comparez avec le code assembleur du départ.

### Exercice n° 7

1. Créez un fichier "main.c" contenant le code suivant :

```
int main() {
   return sum(10);
}
```

[1] Le niveau d'optimisation par défaut étant `O0`, l'option `-O0` peut donc être omise.

2. Générez l'exécutable de la manière suivante :

```
$ gcc -c -Og -o sum.o sum.c
$ gcc -c -Og -o main.o main.c
$ gcc -o sum sum.o main.o
```

3. Exécutez le programme `sum`. Que se passe-t-il ? Comment vérifier la valeur retournée par le programme en utilisant une commande Unix ?
4. Quelle est la raison de l'avertissement lors de la production de "main.o" ? Corrigez le problème.
5. Relevez la taille des fichiers objets ainsi que celle du programme exécutable obtenu. La taille de l'exécutable correspond-elle à la somme des tailles des deux fichiers objets ? Quelle peut en être la raison selon vous ?
6. Désassemblez le code exécutable (`objdump -d sum`). Retrouvez-vous dans les grandes lignes le code assembleur de la fonction `sum()` ?

Le visualiseur hexadécimal `xxd` peut aussi servir si l'on n'a pas besoin de désassembler le programme : `xxd sum | more`.

## 2 Livrable

Avant la date limite, déposez sur la plateforme Moodle le fichier "rapport-tp05-prenom_nom.md" qui contiendra votre compte-rendu.

## 3 Résumé

Dans ce TP vous avez mis en oeuvre la commande `gcc` ainsi qu'une partie de ses nombreuses options de manière à réaliser les différentes étapes de la chaîne de compilation. Vous avez aussi découvert les commandes sous-jacentes qui réalisent ces étapes, notamment les commandes **`cpp`** (le préprocesseur), **`cc1`** (le compilateur), **`as`** (l'assembleur) et **`ld`** (l'éditeur de liens).

De même, vous ne devriez plus à présent vous passer des options **`-Wall`** et **`-Wextra`**, ainsi que **`-ansi`** et **`-pedantic`**[2].

[2] Nous verrons dans le TP de déverminage qu'il est aussi recommandé d'utiliser l'option **`-g`** lors de la phase de mise au point.

Enfin, vous avez découvert la commande `objdump` et appris qu'il était possible de connaître la valeur retournée par un programme[3] à l'aide de la commande du ***shell*** : `echo $?`.

[3] Et oui, ce n'est pas toujours la valeur 0 qui est retournée.