Le compilateur GCC

Présentation

GCC (*GNU Compiler Collection*) est un ensemble de compilateurs créés par le projet GNU. GCC est un logiciel libre capable de compiler divers langages de programmation, dont les langages C (`gcc`) et C++ (`g++`).

Lien : gcc.gnu.org

> GCC est aujourd'hui le compilateur le plus utilisé dans la communauté des logiciels libres et il est le compilateur de nombreux systèmes d'exploitation dont GNU/Linux.

Alternative à GCC : **Clang** est un compilateur pour les langages de programmation C, C++ et Objective-C. Son interface de bas niveau utilise les bibliothèques LLVM pour la compilation. C'est un logiciel libre issu d'un projet de recherche universitaire.

Clang est aujourd'hui maintenu par une large communauté, dont de nombreux employés de Apple, Google, ARM ou Mozilla.

> Ce document se focalise sur le compilateur C++ `g++` .

La syntaxe de base utilisée par le compilateur `g++` est :

```
$ g++ fichier-source.cpp -o binaire
```

`g++` possède plus de 1500 options !

Petit tour d'horizon des options usuelles.

> Le manuel : `man g++`

La version

`g++ --version`

Sans options

Il est possible d'utiliser `g++` sans aucune option. Dans ce cas, `g++` produit un exécutable `a.out` (ou `a.exe` sous Windows) par défaut :

```
$ g++ add.cpp

$ ls -l a.out
-rwxrwxr-x 1 tv tv 8168 sept.  5 09:21 a.out

$ ./a.out
```

Le fichier `a.out` est ce que l'on appelle un **binaire** "exécutable".

> À l'origine, le premier programme assembleur sur Unix générait un fichier nommé `a.out` (*assembler output*), qui était un format de fichier utilisé dans les premières versions d'Unix. Le nom est resté utilisé par certains compilateurs et éditeurs de liens lorsqu'aucun nom de fichier de sortie n'est précisé, même si cet exécutable n'est pas au format *a.out*.

Notion d'exécutable

Un fichier est dit "exécutable" par le système d'exploitation lorsqu'il possède :

- le droit `x` qui autorisera son exécution
- un contenu "exécutable" : soit des instructions compilées pour un "processeur" (physique ou virtuel) soit des instructions interprétées par un autre programme (on parle alors de script)

Ici `a.out` est un exécutable qui contient du code machine (des instructions exécutées par un microprocesseur sur une plateforme cible) :

```
$ objdump -d a.out

Déassemblage de la section .init :

00000000000004b8 <_init>:
 4b8:   48 83 ec 08             sub    $0x8,%rsp
 4bc:   48 8b 05 25 0b 20 00    mov    0x200b25(%rip),%rax        # 200fe8 <__gmon_sta
 4c3:   48 85 c0                test   %rax,%rax
 4c6:   74 02                   je     4ca <_init+0x12>
 4c8:   ff d0                   callq  *%rax
 4ca:   48 83 c4 08             add    $0x8,%rsp
```

```
 4ce:   c3                      retq
...
00000000000005fa <main>:
 5fa:   55                      push   %rbp
 5fb:   48 89 e5                mov    %rsp,%rbp
 5fe:   c7 45 f4 01 00 00 00    movl   $0x1,-0xc(%rbp)
 605:   c7 45 f8 02 00 00 00    movl   $0x2,-0x8(%rbp)
 60c:   8b 55 f4                mov    -0xc(%rbp),%edx
 60f:   8b 45 f8                mov    -0x8(%rbp),%eax
 612:   01 d0                   add    %edx,%eax
 614:   89 45 fc                mov    %eax,-0x4(%rbp)
 617:   b8 00 00 00 00          mov    $0x0,%eax
 61c:   5d                      pop    %rbp
 61d:   c3                      retq
 61e:   66 90                   xchg   %ax,%ax
...
```

> Dans l'affichage précédent, la partie droite correspond à la "vision" en langage Assembleur du code machine (partie gauche). Chaque ligne réprésente une instruction "machine". Les instructions "machine" exécutées par le processeur sont très simples : des lectures/écritures dans la mémoire (`mov` , `push` , `pop` , ...), des opérations arithmétiques (`sub` , `add` , ...) et logiques (`and` , `xor` , ...), des tests avec branchements (`test` , `je` , ...) et des appels de sous-programme (`call` et `ret`).

Evidemment, le contenu du fichier est "binaire" :

```
$ hexdump -C a.out
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  03 00 3e 00 01 00 00 00  f0 04 00 00 00 00 00 00  |..>.............|
00000020  40 00 00 00 00 00 00 00  e8 18 00 00 00 00 00 00  |@...............|
...
```

> Les quatre premiers octets du fichier (ici `.ELF`) représente le **nombre magique** qui permet de désigner le format du fichier.

Même s'il contient des informations sous forme de chaînes de caractères :

```
$ strings a.out
/lib64/ld-linux-x86-64.so.2
libc.so.6
__cxa_finalize
__libc_start_main
GLIBC_2.2.5
_ITM_deregisterTMCloneTable
__gmon_start__
_ITM_registerTMCloneTable
AWAVI
AUATL
[]A\A]A^A_
;*3$"
```

```
GCC: (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0
...
```

On peut supprimer certaines de ces informations inutiles pour l'exécution et donc réduire sa taille :

```
$ ls -l a.out

$ strip a.out

$ ls -l a.out
-rwxrwxr-x 1 tv tv 6056 sept.  5 09:31 a.out
```

Comme tout fichier "binaire", il possède un "format" (c'est-à-dire une structure comprenant notamment des méta-données) :

```
$ file a.out
a.out: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, int
```

Ici, l'exécutable `a.out` possède le format ELF (*Executable and Linkable Format*) pour une architecture `x86-64` et une plateforme `GNU/Linux 3.2.0`. Il n'est donc pas portable vers d'autres systèmes comme Windows par exemple.

Cet exécutable `a.out` a d'autres dépendances, notamment les bibliothèques dynamiques :

```
$ ldd a.out
    linux-vdso.so.1 (0x00007ffd4add6000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f3aa1499000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f3aa1a8c000)
```

Les formats pour les autres plateformes sont :

- sous Windows : format PE (*Portable Executable*)
- sous Mac OS X : Mach-O

Les phases de fabrication

Tout d'abord, il faut savoir que `g++` regroupe dans une seule commande plusieurs outils distincts qui correspondent à des phases précises du processus de fabrication d'un exécutable (ou d'une bibliothèque) :

- le préprocesseur (pré-compilation) `g++ -E`
- le compilateur `g++ -c`
- l'assembleur `g++ -S`
- l'éditeur de liens

> Les explications sur ces différentes phases ont déjà été abordées dans ce *post* : Comment écrire son premier programme en C++ ?.

Il est possible de demander à g++ de décomposer "verbalement" (*verbose*) toutes ces étapes avec les options `-v -save-temps`. Par abus de langage, on nomme souvent `g++` comme le compilateur bien que ce ne soit qu'une simple phase du processus de fabrication.

Le fichier de sortie : `-o`

La première option à connaître (et à placer toute à la fin de la commande) est `-o <fichier>`. Elle permet de préciser le nom de fichier qui contiendra le résultat produit par la commande : fichier exécutable, fichier objet, fichier assembleur ou du code C précompilé.

Si l'option `-o` n'est pas spécifié, une valeur par défaut sera utilisée : `a.out` pour un fichier exécutable, `<fichier>.o` pour un fichier objet, `<fichier>.s` pour un fichier assembleur, `<fichier>.extension.gch` pour un fichier d'en-tête précompilé.

> Les fichiers `.gch` sont des fichiers en-tête précompilés. Ils doivent parfois être supprimés manuellement pour déclencher une re-fabrication avec `make`. Lien : https://en.wikipedia.org/wiki/Precompiled_header

Exemples :

```
$ g++ add.cpp -o add

$ ls -l add
-rwxrwxr-x 1 tv tv 8168 sept.  5 09:53 add
```

ou :

```
$ g++ -c add.cpp -o add.obj
$ g++ add.obj -o add.out

$ ls -l add.out
-rwxrwxr-x 1 tv tv 8168 sept.  5 09:57 add.out
```

La compilation : `-c`

Lorsqu'on pratique la fabrication modulaire (multi-fichiers), il est indispensable d'utiliser l'option `-c` qui déclenche la compilation seule du fichier source passé en argument de la commande `g++`. Une compilation produit un fichier objet avec l'extension `.o`. Il est génralement inutile d'ajouter l'option `-o` car le comportement par défaut convient si l'on souhaite conserver le même nom de fichier.

Exemples :

```
$ g++ -c add.cpp

$ ls -l add.o
-rw-rw-r-- 1 tv tv 1248 sept.  5 09:52 add.o
```

ou :

```
$ g++ -c add.cpp -o toto.obj

$ ls -l toto.obj
-rw-rw-r-- 1 tv tv 1248 sept.  5 09:54 toto.obj
```

Les avertissements : `-W` et `-Werror`

Le compilateur fait une distinction entre les erreurs (qui l'empêche de produire le résultat demandé) et les avertissements (*warning*) qui nous sont destinés lorsque quelque chose lui semble étrange. L'option `-W<format>` permettra de sélectionner ou pas les messages d'avertissement détectés par `g++`. Les *warnings* ne concernent que la phase de compilation (`-c`).

L'option `-Wall` les active tous. C'est une bonne habitude car le débutant a trop tendance à les ignorer contrairement aux développeurs experts. Normalement, une fabrication ne doit produire aucun message d'avertissement et il faut donc les corriger avant de pouvoir exploiter le fichier produit en toute sérénité. Il est possible parfois de désactiver le *warning* avec une option du type `-fno-xxx` mais ce n'est pas conseillé.

Exemple :

```
$ g++ -Wall -c add.cpp
add.cpp: In function ‘int main()’:
add.cpp:3:15: warning: variable ‘c’ set but not used [-Wunused-but-set-variable]
     int a, b, c;
```

> On conseille fortement de transformer les *warnings* en erreurs avec l'option `-Werror`, puis de tous les corriger.

Le choix de la norme du langage : `-std`

Le langage C++ est aujourd'hui normalisé par l'ISO. Sa première normalisation date de 1998 (ISO/CEI 14882:1998), ensuite amendée par l'erratum technique de 2003 (ISO/CEI 14882:2003). Une importante mise à jour a été ratifiée et publiée par l'ISO en septembre 2011 sous le nom de ISO/IEC 14882:2011, ou C++11. Depuis, des mises à jour sont publiées régulièrement : en 2014 (ISO/CEI 14882:2014 ou C++14) puis en 2017 (ISO/CEI 14882:2017 ou C++17).

Ces différentes évolutions du langage C++ concernent aussi bien le langage initial que la bibliothèque standard.

Il est possible de choisir le standard à appliquer par `g++` avec l'option `-std=`. Si l'option n'est pas précisée, `g++` applique un choix par défaut qui dépend de la version de `g++`. Par exemple avec la version 7.5.0, c'est l'option `-std=c++14` qui est appliquée par défaut. Il est habituel d'utiliser l'option `-std=c++11` qui est un standard important et répandu.

Lire : http://tvaira.free.fr/dev/cours/c++-moderne.html et https://github.com/tvaira/cpp-moderne

Exemples :

```
$ g++ -std=c++11 add.cpp
```

ou :

```
$ g++ -std=c++17 add.cpp
```

Les chemins des fichiers d'en-tête (*header*) : `-I`

Il est possible (autant de fois que l'on veut) de préciser à `g++` des chemins qui contiennent des fichiers d'en-tête (*header*) dont on a besoin avec l'option `-I<chemin>`.

Exemple :

```
$ g++ -I./include -c add.cpp
```

Les bibliothèques : `-l` et `-L`

Pendant la phase d'édition de liens, `g++` a besoin de connaître les bibliothèques à associer au programme exécutable à produire.

L'option `-l<bibliothèque>` informe l'éditeur de liens que `<bibliothèque>` est nécessaire pour ce programme. Cette option respecte une convention : le fichier de bibliothèque doit se nommer `lib<bibliothèque>.so` pour GNU/Linux. Dans l'option `-l`, on n'utilise jamais le préfixe `lib` ni l'extension (ici `.so`) du fichier de bibliothèque. L'extension `.so` (*shared object*) est celle utilisée sous GNU/Linux pour les bibliothèques dynamiques. C'est l'équivalent des `.dll` (*dynamic link library*) sous Windows. La convention est identique pour les bibliothèques statiques qui ont elles habituellement les extensions `.a` ou `.lib`.

L'option `-L<chemin>` précisera un chemin où se trouve des bibliothèques. On évitera d'utiliser des chemins absolus pour rendre la fabrication portable quelque soit la configuration de la machine.

Dans le cas d'une édition de liens dynamique (c'est le choix par défaut actuellement), les bibliothèques dynamiques seront chargées en mémoire (et donc utilisées par le programme) au moment de l'exécution du programme. Les erreurs de dépendance (présence, version, ...) ne seront signalées qu'à ce moment là.

Exemple :

```
$ g++ -L./lib -lm -lpthread -c add.cpp -o add.out
```

Le débogage (*debug*) : `-g`

Un débogueur (ou débugueur, de l'anglais *debugger*) est un logiciel qui aide un développeur à analyser les défauts (*bugs*) d'un programme.

Pour cela, il permet d'exécuter le programme pas-à-pas, d'afficher la valeur des variables à tout moment, de mettre en place des points d'arrêt sur des conditions ou sur des lignes du programme ...

Le programme à débuguer est exécuté à travers le débogueur et s'exécute normalement. Le débogueur offre alors au programmeur la possibilité d'observer et de contrôler l'exécution du programme.

Les programmes C/C++ doivent obligatoirement être compilés avec l'option `-g` pour pouvoir être débuggés par gdb.

Lire : GNU Debugger gdb

Exemples :

```
$ g++ -g add.cpp

$ ls -l a.out
-rwxrwxr-x 1 tv tv 9088 sept.  5 10:04 a.out

$ file a.out
a.out: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, int

$ gdb -q ./a.out
Reading symbols from ./a.out...done.
(gdb) list
1       int main()
2       {
3           int a, b, c;
4           a = 1;
5           b = 2;
6           c = a + b;
7           return 0;
8       }
(gdb) start
Temporary breakpoint 1 at 0x5fe: file add.cpp, line 4.
Starting program: /home/tv/Téléchargements/a.out
warning: the debug information found in "/lib64/ld-2.27.so" does not match "/lib64/ld-

Temporary breakpoint 1, main () at add.cpp:4
4           a = 1;
(gdb) step
5           b = 2;
(gdb)
...
```

Les étiquettes : `-D`

L'option `-D` est strictement identique à la directive de pré-compilation `#define`

Cela permet donc de définir des étiquettes. Par exemple, l'option `-DDEBUG` (il n'y a pas d'espace) fera la même chose que `#define DEBUG` placé dans un fichier d'en-tête `.h`.

Voir aussi : l'option `-DNDEBUG` et `assert()`.

> C'est une option très utilisée.

L'optimisation : `-O`

Cette option contrôle les différents types d'optimisations.

Par défaut, le but du compilateur est de réduire le coût de la compilation et de faire en sorte que le débogage produise les résultats attendus.

L'activation de l'optimisation obligera le compilateur à tenter d'améliorer les performances et/ou la taille du code au détriment du temps de compilation et éventuellement de la capacité pour déboguer le programme.

Il est possible d'indiquer un niveau (*level*) d'optimisation avec l'option `Olevel` (de `1` à `3`, le niveau `0` désactive l'optimisation).

On peut spécifiquement chercher à optimiser :

- la taille avec `-Os`
- les performances en vitesse avec `-Ofast`
