# Utilisation de la collection de compilateurs de GNU

## Gazette Linux n°120 — Novembre 2005


### Vinayak Hegde





**Article paru dans le n°120 de la Gazette Linux de novembre 2005.**
**Traduction française par Sébastien Duburque <SebastienDuburque CHEZ gmail POINT com>.**
**Réécriture, correction et relecture de la traduction française par Joëlle Cornavin <jcornavi CHEZ club TIRET internet POINT fr>.**



## Table des matières



## 1. Introduction à GCC

Le compilateur C de GNU fait partie intégrante du système GNU et a été initialement écrit par Richard M. Stallman. Au départ, il ne compilait que du code C. Plus tard, un groupe de volontaires à commencé à en assurer la maintenance, et GCC a réussi à obtenir la prise en charge de différents langages tels que

C++, Fortran Ada et Java. Il a été ensuite renommé en GNU Compiler Collection) pour signifier ce changement. Dans cet article, nous étudierons principalement le compilateur du langage C.

GCC n'est pas seulement disponible sous Linux mais aussi sous d'autres systèmes Unix, tels que FreeBSD, NetBSD, OpenBSD et même sous Windows via Cygwin, MingW32 et Microsoft Services pour Unix. Il prend en charge une grande variété de plates-formes, telles que les architectures Intel x86, AMD x86-64, Alpha et SPARC. En raison de cette polyvalence, on l'utilise souvent pour produire du code de compilation croisée pour différentes architectures. Du fait que le code source de GCC est disponible et modulaire, on peut facilement le modifier pour créer des binaires pour des plates-formes nouvelles ou peu connues, telles que celles qu'emploient les systèmes embarqués.

# 2. Options élémentaires de compilation

Si GCC est disponible sur votre système, vous pouvez saisir la commande suivante pour voir avec quelles options il a été compilé.

## 2.1. Commande n°1 : spécifications et fonctionnalité prise en charge de GCC

Cette commande donne beaucoup d'informations sur GCC. Vous pouvez constater que comme cette version prend en charge le modèle de *threading* posix, vous pouvez compiler des applications multi-*threaded* avec. Elle peut aussi compiler du code écrit en C, C++, Fortran-77, Objective C, Java et Ada. Notez que le chemin d'inclusion C++ est également spécifié et que le code Java peut être compilé en binaires natifs avec libgcj.

Écrivons un petit programme en C (outils/lg120-D/helloworld.c.txt) avec un fichier d'en-tête (outils/lg120-D/helloworld.h.txt) pour voir les différentes options de compilation que gère GCC.

Voici le fichier C :

## 2.2. Commande n°2 : création d'un fichier objet

Pour compiler le programme helloworld en un fichier objet, on peut utiliser la commande suivante :

## 2.3. Commande n°3 : création d'un exécutable

D'après les sorties ci-dessus, nous pouvons constater que **gcc** appelle **cc1** qui est le compilateur C proprement dit pour générer un fichier d'assembleur appelé `ccHmbDAJ.s`. C'est un nom choisi aléatoirement et ce fichier est supprimé dès que la compilation est terminée. Il est également possible de voir dans quel ordre sont recherchés les différents chemins des fichiers d'inclusion. Nous pouvons modifier les chemins de recherche des fichiers d'inclusion à l'aide de l'option `-I` et les chemins de recherche des bibliothèques via l'option `-L`. Consultez les pages info (**$ info gcc**) pour plus d'informations sur ces options. Le fichier d'assembleur temporaire créé est ensuite passé sur l'assembleur GNU (**as**) qui traite le fichier et génère du code binaire pour cette plate-forme particulière. Le processus s'arrête ici pour le fichier objet (commande n°1).

Lors de la création d'un fichier exécutable, une étape supplémentaire entre en jeu —: la liaison. D'après la sortie de la commande n°2, nous pouvons voir que le fichier est lié dynamiquement avec les bibliothèques (notez l'usage de l'option `-L` ici également). collect2 est un utilitaire qui configure les routines d'initialisation et appelle éventuellement **ld** pour effectuer la liaison afin de créer l'exécutable.

# 3. Rôle du préprocesseur

Le préprocesseur est une partie importante du compilateur C. Toutes les directives du préprocesseur commencent par un caractère `#` (dièse). Il traite les différentes directives du préprocesseur telles que `#define`, `#include`, `#ifdef`, `#pragma` et `#undef`. Comme le nom le suggère, le préprocesseur tourne avant que la compilation du programme ne commence et traite les diverses directives pour produire du code prêt à être compilé par le compilateur C. Il est possible de définir des macros assez complexes à l'aide des directives, ce qui peut rendre le code plus lisible et réduire la complexité. Cependant, il se peut que les macros complexes ne s'étendent pas comme nous penserions qu'elles le fassent. De plus, si certains des fichiers d'inclusion (*include files*) ont le même nom, il est possible que le mauvais fichier soit choisi et cause des erreurs de compilation ou un comportement bizarre dans l'exécutable. Dans de tels cas, on peut utiliser l'option `-E` de façon à obtenir la sortie du préprocesseur telle que le compilateur la voit. Nous pouvons réutiliser l'exemple ci-dessus pour voir la sortie de préprocesseur que produit le compilateur.

## 3.1. Commande n°4 : sortie du préprocesseur

## 3.2. Commande n°5 : macros étendues (`#define`)

La commande n°4 produira un long fichier de préprocesseur avec tous les fichiers inclus et toutes les macros étendues. Vous pouvez ouvrir ce fichier dans votre éditeur favori et y jeter un coup d'œil. C'est le source C que le compilateur C examine. Quand j'ai lancé la commande ci-dessus sur mon ordinateur de bureau, j'ai obtenu 455 lignes de sortie, en excluant les espaces. La commande n°5 montre toutes les macros #define définies après qu'elles ont été triées. Il est également possible de définir des macros sur la ligne de commande de la compilation. Regardez par exemple la sortie de la commande n°2, où `__GNUC__`, `__GNUC_MINOR__` et `__GNUC_PATCHLEVEL__` sont toutes définies comme la commande n°3 puisque la version de GCC utilisée pour la compilation est GCC 3.3.3.

# 4. Génération d'une sortie en langage assembleur

GCC convertit le code C en langage assembleur avant de le convertir en code binaire. Dans certains cas, vous pourriez être amené à examiner le code généré ou à le modifier pour des raisons de performances avant de le convertir en code binaire. Vous pouvez le faire à l'aide de la commande suivante :

La sortie produite est comme suit :

D'après la sortie ci-dessus, nous pouvons voir que ***hello*** est défini en tant que chaîne de caractères ***helloworld***. C'est une donnée en lecture seule car nous l'avons définie comme `static`. `main` est la seule fonction globale. La section `.LC0` montre les paramètres pour `printf`. Ceux-ci sont alors placés sur la pile avant que `printf` ne soit appelée dans la boucle principale (`.L5`). La section `.L2` contient le code pour vérifier les conditions de la boucle **for**. La section `.L3` contient les routines de nettoyage après que la fonction est terminée. La sortie en langage assembleur humainement lisible générée peut être changée avant d'être compilée en code binaire à l'aide de **as** (l'assembleur de GNU) puis liée avec les bibliothèques.

# 5. Conformité et options d'avertissement

GCC possède ses propres extensions pour le C standard. Beaucoup de programmes GNU ainsi que d'autres logiciels, dont le noyau Linux. Il se peut que ces extensions ne soient pas disponibles avec d'autres compilateurs sur d'autres plates-formes. Ainsi, si vous voulez écrire du code portable, vous serez peut-être amené à employer l'option `-ansi`. Utiliser cette option associée à l'option `-pedantic` garantira que tout code non conforme à la norme C ISO sera signalé par un avertissement. De plus, il nous est possible de spécifier le standard auquel nous souhaitons adhérer via l'option `-std=option`. Les standards que cette option prend en charge incluent la norme C89 ISO (`-std=c89`), la norme C99 ISO (`-std=c99`) et la norme C++ ISO (`-std=c++98`).

Il est en outre toujours judicieux d'activer quelques avertissements courants, à l'aide de l'option `-Wall`. Mais `-Wall` n'active pas tous les messages d'avertissements. Ce n'est donc pas un terme approprié. Quelques-unes des autres options d'avertissement que vous pourriez être amené à activer sont : `-Wstrict-prototypes` et `-Wmissing-prototypes` ((avertissement si les prototypes ne sont pas définis ou définis de façon incorrecte), `-Werror` (qui transforme tous les avertissements en erreurs) et `-wunreachable-code` (si le compilateur constate qu'un bloc de code ne s'éxécutera jamais).

# 6. Génération des dépendances d'un `Makefile`

**make** est un outil de construction automatisé auquel on fait appel pour construire un grand nombre de fichiers dans un projet C. Il fera l'objet d'un article ultérieur de cette série. Si vous avez (par exemple) 1 000 fichiers dans un projet et que vous changez juste un ou deux fichiers pour corriger un bogue, vous n'avez pas besoin de recompiler tout le projet. Vous pouvez indiquer quels sont les fichiers affectés par les changements en spécifiant les dépendances, et seuls ces fichiers seront recompilés. Vous pouvez utiliser GCC pour générer ces lignes de dépendances. Jetez un coup d'œil à l'exemple ci-dessous :

Cette habile petite astuce peut vous faire gagner beaucoup de temps quand vous travaillez avec une date limite.

# 7. Utilisation de code de bibliothèque

Si vous développez une bibliothèque à l'usage d'autres développeurs, vous devez employer l'option `-fpic` pour générer du PIC *Position Independent Code*, code indépendant de la position). Quand un exécutable est créé, certains *offsets* de fonctions et de données sont codés en dur dedans. Pour une

bibliothèque, ce n'est manifestement pas une option puisque le code de la bibliothèque doit être indépendant des *offsets* d'emplacements codés en dur — le code de bibliothèque sera à terme lié dans l'exécutable (dynamiquement ou statiquement). De plus, si vous avez un composant qui doit être lié à de multiples exécutables, il faut employer l'option `-shared` de **gcc**. Cette dernière est le plus souvent associée à l'option `-fpic` pour créer des bibliothèques partagées.

Sur la plupart des systèmes, le comportement par défaut de **gcc** est de lier dynamiquement, ce qui peut créer des problèmes si vous ne voulez pas distribuer la bibliothèque partagée en même temps que l'exécutable. En outre, vous pouvez être dans une situation où la bibliothèque partagée que vous avez utilisée sur votre système n'est pas rapidement disponible. Dans ce cas, nous pouvons statiquement lier l'exécutable, de façon à ce que le code de la bibliothèque n'ait pas à être fourni séparément. Utilisez cependant cette option avec prudence car elle augmentera légèrement la taille de l'exécutable. L'option de commande pour lier statiquement la sortie de **gcc** est (sans surprise) `-static`.

# 8. Conclusion

Dans cet article, nous avons fait un petit tour d"horizon de la manière dont **gcc** peut être utilisé pour générer des binaires et les diverses étapes que le code C parcourt avant d'être converti en code binaire. Dans le prochain article de cette série, nous verrons comment optimiser le code généré pour une plate-forme particulière, ainsi que les options pour générer des binaires de débogage à employer avec **gdb**.

Vinayak Hegde travaille actuellement pour *Akamai Technologies Inc*. Ses premiers pas sous Linux datent de 1997 et il n'est jamais revenu en arrière depuis. Il s'intéresse aux réseaux informatiques à grande échelle, aux systèmes informatiques distribués et aux langages de programmation. Pendant son temps libre inexistant, il aime faire de la randonnée, écouter de la musique et lire. Il maintient également un blog (http://www.livejournal.com/users/vinayakh) qu'il met à jour par intermittence.