LinuxFocus article number 229
http://linuxfocus.org

par Lorne Bailey
<sherm_pbody/at/yahoo.com>

L´auteur:

Lorne vit à Chicago et travaille comme consultant spécialisé dans les transferts de données entre bases Oracle. Comme il a choisi de programmer en environnement Unix exclusivement, Lorne a pu ainsi éviter "l'enfer des DLL". Il travaille actuellement à sa maîtrise en informatique.

Traduit en Français par:
Georges Tarbouriech
<georges.t/at/linuxfocus.org>

GCC - la base de tout

Résumé:

Cet article implique que vous possédiez les bases du langage C et vous montrera comment utiliser le compilateur gcc. Nous vérifierons que vous puissiez invoquer le compilateur depuis la ligne de commande pour du simple code source C. Nous verrons ensuite ce qui se passe réellement et comment vous pouvez contrôler la compilation de vos programmes. Nous survolerons également l'utilisation d'un débogueur.

Les règles de GCC

Pouvez-vous imaginer la compilation de logiciel Libre avec un compilateur propriétaire dont vous ne possédez pas les sources ? Comment pourriez-vous savoir ce qui entre dans votre exécutable ? Il pourrait contenir tous les types de "back door" ou de chevaux de Troie. Ken Thompson, dans l'un des plus gros "coups" de l'histoire, avait écrit un compilateur qui introduisait un "back door" dans le programme de "login" et perpétuait le cheval de Troie quand le compilateur réalisait qu'il se compilait lui-même. Lisez sa description de ce grand classique ici. Par chance, nous avons gcc. Chaque fois que vous tapez

`configure; make; make install` , gcc fait un gros ménage qui ne se voit pas. Comment allons-nous faire travailler gcc ? Nous allons commencer à écrire un jeu de cartes, mais nous n'écrirons que ce qui est nécessaire à la démonstration des fonctionnalités du compilateur. Comme nous partons de zéro, il faut comprendre le processus de compilation pour savoir ce qui doit être fait, dans quel ordre, pour créer un exécutable. Nous allons voir comment un programme en C se compile et les options qui permettent à gcc de faire ce qu'on attend de lui. Les différentes étapes (et les outils qui vont avec) sont Précompilation (gcc -E), Compilation (gcc), Assemblage (as), et Lien (ld).

Pour commencer...

Tout d'abord, la première chose à savoir c'est comment invoquer le compilateur. C'est en fait, très simple. Nous allons commencer par le classique premier programme en C. (Les vieux routiers devront me pardonner).

```
#include <stdio.h>

int main()
{
  printf("Hello World!\n");
}
```

Enregistrez ce fichier sous le nom `game.c`. Vous pouvez le compiler depuis la ligne de commande en tapant :

```
gcc game.c
```

Par défaut, le compilateur crée un exécutable nommé `a.out`. Vous pouvez l'exécuter en tapant :

```
a.out
Hello World
```

Chaque fois que vous compilez un programme, le nouveau `a.out` écrase le précédent. Vous ne pouvez pas dire quel est le programme ayant créé le `a.out` actuel. Nous pouvons résoudre ce problème en disant à gcc que nous voulons nommer l'exécutable grâce à l'option `-o`. Nous appellerons ce programme `game`, même si on peut lui donner n'importe quel nom, le C n'ayant pas les mêmes restrictions de nommage que Java.

```
gcc -o game game.c

game
Hello World
```

A ce stade, nous sommes très loin d'avoir un programme utile. Si vous pensez que ça ne sert à rien, considérez plutôt le fait d'avoir un programme qui se compile et s'exécute. Au fur et à mesure que nous ajouterons des fonctionnalités à ce programme, nous devrons vérifier qu'il est toujours capable de s'exécuter. Un programmeur débutant veut souvent écrire 1000 lignes de code et résoudre tous les problèmes en même temps. Personne, je dis bien personne, n'est capable de faire ça. Vous créez un petit programme qui fonctionne, vous faites des modifications et vous l'exécutez de nouveau. Ceci limite le

nombre d’erreurs à corriger d’un coup. De plus, vous savez exactement ce que vous venez de changer et qui ne fonctionne pas, vous savez donc sur quoi vous concentrer. Ceci vous empêche de créer quelque chose dont **vous** pensez que ça devrait fonctionner, qui est peut-être même compilable, mais qui ne deviendra jamais un exécutable. Rappelez-vous, ce n’est pas parce qu’un programme se compile qu’il est fonctionnel.

Notre prochaine étape consiste à créer un fichier entête (header) pour notre jeu. Un fichier entête permet de regrouper des types de données et des déclarations de fonctions en un seul endroit. Cela permet de garantir que les structures de données sont bien définies afin que chaque morceau du programme voit les choses exactement de la même manière.

```
#ifndef DECK_H
#define DECK_H

#define DECKSIZE 52

typedef struct deck_t
{
  int card[DECKSIZE];
  /* nombre de cartes utilisées */
  int dealt;
}deck_t;

#endif /* DECK_H */
```

Enregistrez ce fichier sous le nom `deck.h`. Seuls les fichiers `.c` sont compilés, nous devons donc modifier game.c. A la ligne 2 de game.c, écrivez `#include "deck.h"`. A la ligne 5, écrivez `deck_tdeck;`. Pour être certain que rien n’a été altéré, recompilons-le.

```
gcc -o game game.c
```

Pas d’erreurs, c’est parfait. S’il ne se compile pas, corrigez-le jusqu’à ce qu’il puisse.

Précompilation

Comment le compilateur connaît-il le type de `deck_t` ? Parce que pendant la précompilation, il copie effectivement le fichier "deck.h" dans le fichier "game.c". Les directives de précompilation dans le code source sont préfixées par un "#". Vous pouvez invoquer le précompilateur par le frontal gcc avec l’option `-E`.

```
gcc -E -o game_precompile.txt game.c
wc -l game_precompile.txt
  3199 game_precompile.txt
```

Pratiquement 3200 lignes ! La plupart viennent du fichier include `stdio.h`, mais si vous regardez de plus près, nos déclarations y sont aussi. Si vous ne fournissez pas un nom de fichier de sortie par l’option `-o`, tout sera écrit sur la console. Le processus de précompilation donne une plus grande flexibilité dans le code en accomplissant trois tâches essentielles.

1. Il copie les fichiers "#include" dans le fichier source à compiler.

2. Il remplace les textes "#define" par leur valeur réelle.
3. Il remplace les macros lorsqu'elles sont appelées.

Cela vous permet d'avoir des constantes nommées (par exemple DECKSIZE représente le nombre de cartes dans un jeu) utilisées par la totalité du source, définies en un seul endroit et mises à jour partout chaque fois que la valeur change. En pratique, vous n'utilisez presque jamais l'option `-E` toute seule, mais vous lui laissez envoyer sa sortie au compilateur.

Compilation

Comme étape intermédiaire, gcc traduit votre code en langage Assembleur. Pour ce faire, il doit comprendre le but que vous cherchez à atteindre en analysant votre code. Si vous avez fait une erreur de syntaxe, il vous avertira et la compilation échouera. Parfois, les utilisateurs confondent cette étape avec le processus complet. Pourtant, gcc a encore beaucoup de travail à accomplir.

Assemblage

`as` transforme le code Assembleur en code objet. Le code objet ne peut pas encore fonctionner avec le processeur, mais il n'en est pas loin. L'option `-c` du compilateur convertit un fichier .c en un fichier objet avec une extension .o. Si nous tapons

```
gcc -c game.c
```

nous créons automatiquement un fichier nommé game.o. Nous mettons ici le doigt sur un point important. Nous pouvons prendre n'importe quel fichier .c et le transformer en fichier objet. Comme nous le verrons ci-dessous, nous pouvons alors combiner plusieurs fichiers objet dans un même exécutable pendant la phase de lien. Continuons avec notre exemple. Puisque nous programmons un jeu de cartes et que nous l'avons défini en tant que `deck_t`, nous allons écrire une fonction pour battre les cartes. Cette fonction génère un pointeur sur un type de jeu et lui donne un ensemble de valeurs aléatoires pour les différentes cartes. Le tableau "drawn" permet de mémoriser les cartes déjà utilisées. Ce tableau des membres de DECKSIZE empêche de donner plusieurs fois la même valeur à une carte.

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include "deck.h"

static time_t seed = 0;

void shuffle(deck_t *pdeck)
{
  /* Mémorise les nombres utilisés */
  int drawn[DECKSIZE] = {0};
  int i;

  /* Initialisation aléatoire unique */
  if(0 == seed)
```

```
  {
    seed = time(NULL);
    srand(seed);
  }
  for(i = 0; i < DECKSIZE; i++)
  {
    int value = -1;
    do
    {
      value = rand() % DECKSIZE;
    }
    while(drawn[value] != 0);

    /* marque la valeur comme déjà utilisée */
    drawn[value] = 1;

    /* pour débogage */
    printf("%i\n", value);
    pdeck->card[i] = value;
  }
  pdeck->dealt = 0;
  return;
}
```

Enregistrez ce fichier sous le nom `shuffle.c`. Nous avons ajouté du code de débogage, ainsi, lorsque le programme s'exécute, il écrit les numéros de cartes qu'il génère. Ca n'ajoute rien aux fonctionnalités de notre programme, mais il est essentiel à ce stade de voir ce qui se passe. Comme nous n'en sommes qu'au début de notre jeu, nous n'avons pas d'autre moyen de vérifier que notre fonction effectue bien ce que nous en attendons. Grâce à printf, nous voyons exactement ce qui se produit, ainsi, quand nous passerons à la phase suivante nous saurons que le jeu est correctement battu. Maintenant que nous savons que le fonctionnement est correct, nous pouvons retirer cette ligne de notre code. Cette technique de débogage peut sembler rudimentaire, mais elle fait ce que l'on en attend tout en restant minimaliste. Nous aborderons un débogage plus sophistiqué un peu plus loin.

Remarquons deux choses.

1. Nous passons un paramètre par son adresse, que vous obtenez par le '&' (adresse de) l'opérateur. Ceci passe l'adresse de la variable de la machine à la fonction, lui permettant de changer la variable proprement dite. Il est possible de programmer avec des variables globales, mais elles ne devraient être utilisées que rarement. Les pointeurs sont une part importante du C et il est essentiel de bien les comprendre.
2. Nous utilisons un appel de fonction depuis un nouveau fichier .c. Le système d'exploitation cherche toujours une fonction nommée 'main' et commence toujours l'exécution par là. `shuffle.c` n'a pas de fonction 'main' et par conséquent il ne peut devenir un exécutable autonome. Nous devons le combiner à un autre programme ayant un 'main' et appeler la fonction 'shuffle'.

Tapez la commande

```
gcc -c shuffle.c
```

et vérifiez qu'elle crée bien un nouveau fichier nommé `shuffle.o`. Editez le fichier game.c, et à la ligne 7, après la déclaration de la variable deck_t `deck`, ajoutez la ligne

```
shuffle(&deck);
```

Maintenant, si nous essayons de créer un exécutable comme précédemment, nous obtenons une erreur

```
gcc -o game game.c

/tmp/ccmiHnJX.o: In function ‘main’:
/tmp/ccmiHnJX.o(.text+0xf): undefined reference to ‘shuffle’
collect2: ld returned 1 exit status
```

La compilation a réussi parce que notre syntaxe était correcte. L’étape de lien a échoué parce que nous n’avons pas fourni au compilateur la position de la fonction ’shuffle’. Qu’est-ce que le *lien* et comment dire au compilateur où trouver cette fonction ?

Lien

Le "lieur" (linker), `ld`, prend le code objet précédemment créé par `as` et le transforme en exécutable par la commande

```
gcc -o game game.o shuffle.o
```

Ceci combine les deux objets et crée l’exécutable `game`.

Le lieur trouve la fonction `shuffle` dans l’objet shuffle.o et l’intègre à l’exécutable. La grande "beauté" des fichiers objet vient du fait que si nous souhaitons réutiliser cette fonction, nous devons seulement inclure le fichier "deck.h" et lier le fichier objet `shuffle.o` dans le nouvel exécutable.

La réutilisation de code est presque systématique. Ainsi, nous n’avons pas eu à écrire la fonction `printf` appelée ci-dessus dans un but de débogage, le lieur a trouvé sa définition dans le fichier inclus par `#include <stdlib.h>` et l’a lié au code objet stocké dans la bibliothèque C (/lib/libc.so.6). Ainsi, nous pouvons utiliser les fonctions d’autres programmeurs dont nous savons qu’elles "travaillent" correctement, et nous concentrer sur la résolution de nos propres problèmes. C’est pourquoi les fichiers "headers" ne contiennent normalement que les données et les définitions des fonctions et non les corps des fonctions. Habituellement vous créez des fichiers objet ou des bibliothèques pour que le lieur les intègre dans l’exécutable. Nous pourrions avoir un problème avec notre code parce que nous n’avons mis aucune définition de fonction dans notre fichier entête. Que pouvons-nous faire pour être sûr que tout se passe bien ?

Deux autres options importantes

L’option `-Wall` active tous les avertissements disponibles sur la syntaxe du langage pour nous aider à vérifier que notre code est correct et aussi portable que possible. Lorsque nous utilisons cette option et compilons notre code, nous voyons des choses du style :

```
game.c:9: warning: implicit declaration of function ‘shuffle’
```

Ceci nous informe que nous avons encore un peu de travail à accomplir. Nous devons ajouter une ligne dans un fichier entête dans lequel nous informons le compilateur sur notre fonction `shuffle` de manière à ce qu'il puisse vérifier ce qu'il doit. Ca semble vouloir "couper les cheveux en quatre" (dans le sens de la longueur !), mais en réalité, cela sépare la définition de l'implémentation et permet d'utiliser notre fonction n'importe où simplement en incluant notre nouveau "header" et en le liant à notre code objet. Ajoutons cette simple ligne dans le fichier deck.h.

```
void shuffle(deck_t *pdeck);
```

Voilà qui supprime notre message d'alerte.

Une autre option de compilation répandue est l'optimisation `-O#` (i.e. -O2). Elle indique au compilateur le degré d'optimisation souhaité. Le compilateur possède une pleine escarcelle de trucs permettant de rendre votre code plus rapide. Pour un petit programme comme le notre, vous ne verrez pas grande différence, mais pour des programmes plus gros, l'accélération peut être significative. Vous verrez ça partout, il est donc préférable de savoir de quoi il s'agit.

Débogage

Comme nous le savons, le fait que notre code se compile ne signifie pas qu'il fonctionne comme nous le souhaitons. Vous pouvez vérifier que tous les nombres sont utilisés une seule fois en tapant

```
game | sort - n | less
```

et en contrôlant que rien ne manque. Que faire si un problème apparaît ? Comment regarder sous le capot et trouver l'erreur ?

Vous pouvez contrôler votre code grâce à un débogueur. La plupart des distributions proposent le classique gdb. Si les options de la ligne de commande ne vous passionnent pas, comme c'est le cas pour moi, KDE propose un frontal très agréable avec KDbg. Il en existe d'autres et ils sont très semblables. Pour commencer le débogage, sélectionnez File->Executable et chercher votre programme `game`. Lorsque vous pressez F5 ou sélectionnez Execution->Run dans le menu, vous devriez visualiser la sortie dans une fenêtre. Que se passe-t-il ? Vous ne voyez rien dans la fenêtre. Pas de panique, KDbg n'est pas en cause. Le problème vient du fait que nous n'avons mis aucune information de débogage dans l'exécutable, donc KDbg ne peut pas nous dire ce qui se passe en interne. L'option `-g` du compilateur intègre les informations requises dans les fichiers objet. Vous devez compiler les fichiers objet (extension .o) avec cette option; la commande devient alors :

```
gcc -g -c shuffle.c game.c
gcc -g -o game game.o shuffle.o
```

Ceci insére des repères dans l'exécutable, permettant ainsi à gdb ou KDbg de découvrir ce qui se produit. Le débogage est une technique importante, qui vaut la peine de lui consacrer du temps pour la maîtriser. La manière dont les débogueurs aident les programmeurs vient de la faculté de définir un 'point d'arrêt' dans le code source. Essayez d'en définir un maintenant en cliquant avec le bouton droit sur la ligne contenant l'appel à la fonction `shuffle`. Un petit cercle rouge devrait apparaître après la ligne. Maintenant, lorsque vous pressez F5, le programme arrête son exécution à cette ligne. Pressez F8

pour entrer *dans* la fonction shuffle. Eh oui, nous sommes bien dans le code de `shuffle.c` ! Nous pouvons contrôler l'exécution pas à pas et voir ce qui se passe réellement. Si vous laissez la flèche flotter au-dessus d'une variable locale, vous verrez ce qu'elle contient. Chouette. C'est quand même mieux que les `printf`, non ?

Récapitulation

Cet article a proposé une visite éclair de la compilation et du débogage de programmes C. Nous avons abordé les étapes suivies par le compilateur et les options à passer à gcc pour qu'il fonctionne conformément à notre attente. Nous avons survolé la phase de lien avec les bibliothèques partagées et nous avons terminé par une introduction sur les débogueurs. Savoir ce que vous faites représente un gros travail, mais j'espère que ceci vous aura aidé à démarrer du bon pied. Vous trouverez plus ample information dans les pages `man` et `info` pour `gcc`, `as` et `ld`.

Ecrire du code est ce qui vous instruira le plus. Pour pratiquer, vous pourriez utiliser les simples bases du jeu de cartes de cet article et écrire un jeu de blackjack. Prenez le temps d'apprendre comment utiliser un débogueur. Il est plus facile de commencer avec un outil possédant une interface graphique comme KDbg. Si vous ajoutez seulement peu de fonctionnalités à la fois, vous aboutirez sans même vous en rendre compte. Rappelez-vous, votre application doit toujours être capable de s'exécuter !

Voici quelques petites choses dont vous pourriez avoir besoin pour créer un jeu entier.

- Une définition de joueur (vous pourriez par exemple, définir joueur_t comme vous avez défini deck_t).
- Une fonction qui gère un nombre de cartes donné pour un joueur donné. Pensez à augmenter le nombre 'utilisé' dans le jeu pour savoir d'où gérer la carte suivante. Pensez à mémoriser le nombre de cartes pour un joueur donné.
- Un peu d'interactivité pour demander au joueur s'il veut une autre carte.
- Une fonction affichant les cartes d'un joueur. La *carte* a une valeur de % 13 (s'étendant de 0 à 12), la *couleur* a une valeur de / 13 (de 0 à 3).
- Une fonction pour déterminer la valeur du jeu d'un joueur. Les As ont une valeur de zéro et peuvent valoir 1 ou 11. Les Rois ont une valeur de 12 qui vaut 10.

Liens

- gcc La collection des compilateurs GCC GNU
- gdb Le débogueur GNU
- KDbg Le débogueur graphique de KDE
- Award Winning Compiler Hack Le "coup" de Ken Thompson

2005-01-14, generated by lfparser_pdf version 2.51