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

About the author:

Lorne vive a Chicago a lavora come consulente informatico specializzato nel mettere e togliere dati in un database Oracle. Fin da quando è passato a programmare esclusivamente in ambiente *nix, Lorne ha completamente evitato l'"inferno DLL". Attualmente sta lavorando alla sua tesi in Scienze Informatiche.

GCC – la radice di tutto

Abstract:

Questo articolo presuppone che conosciate le basi del linguaggio C, e vi introdurrà all'uso di gcc come compilatore. Controlleremo che possiate richiamare il compilatore dalla linea di comando con un semplice sorgente in C. Quindi daremo una rapida occhiata a quello che succede e a come potete controllare la compilazione dei vostri programmi. Daremo anche un veloce sguardo all'uso di un debugger.

Le regole di GCC

Riuscite a immaginare di compilare software free con un compilatore proprietario e closed−source? Come sapreste cosa sta succedendo nel vostro eseguibile? Ci potrebbero essere tutti i tipi di backdoor o di trojan. Ken Thompson, in uno dei migliori hack di tutti i tempi, scrisse un compilatore che lasciava una backdoor nel programma 'login' e che replicava il trojan quando il compilatore si accorgeva di ricompilare se stesso. Leggete la descrizione di questo grande classico qui. Fortunatamente abbiamo gcc. Ogni volta che fate `configure; make; make install` gcc fa un bel po' di sollevamento pesi dietro le quinte. Come possiamo fare in modo che gcc lavori per noi? Inizieremo scrivendo un gioco di carte, ma scriveremo solo quello che basta per dimostrare le possibilità del compilatore. Visto che stiamo iniziando da zero, c'è bisogno di capire il processo di compilazione per capire cosa deve essere fatto per creare un eseguibile e in che ordine. Daremo un'occhiata generale a come un programma in C viene compilato e alle opzioni che fanno fare al gcc quello che vogliamo che faccia. I passi (e i tool che li fanno) sono Pre−compilazione (gcc −E), Compilazione (gcc), Assembly (as), and Link (ld).

All'inizio...

Prima di tutto dovremo sapere come richiamare il compilatore. In verità è semplice. Inizieremo col classico primo programma in C. (I veterani dovranno perdonarmi).

```
#include <stdio.h>

int main()

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

salvate questo file come `game.c`. Potete compilarlo dalla linea di comando lanciando:

```
gcc game.c
```

Di default il compilatore C crea un eseguibile chiamato `a.out`. Potete lanciarlo scrivendo:

```
a.out
Hello World
```

Ogni volta che compilate un programma, il nuovo `a.out` sovrascrive il precedente programma. Non sarete in grado di capire quale programma abbia generato l'attuale `a.out`. Possiamo risolvere questo problema dicendo al gcc che nome dare all'eseguibile con l'opzione `-o`. Chiameremo questo programma `game`, anche se potremmo nominarlo come vogliamo, visto che il C non ha le restrizioni di Java sui nomi.

```
gcc -o game game.c

game
Hello World
```

A questo punto siamo abbastanza lontani dall'avere un programma utils. Se pensate che ciò sia un male, dovreste considerare il fatto che abbiamo un programma che si compila e funziona. Man mano che aggiungeremo funzioni a questo programma vogliamo essere sicuri di mantenerlo funzionante. Sembra che ogni programmatore che inizia voglia scrivere 1.000 linee di codice e quindi correggerle tutte in una volta. Nessuno, e intendo nessuno, può fare una cosa del genere. Fate un programma che funziona, fate dei cambiamenti, lo fate funzionare ancora. Questo limita gli errori che dovrete correggere ogni volta. Inoltre saprete esattamente cosa avrete fatto che non funziona, quindi saprete dove concentrarvi. Questo vi evita di creare qualcosa che **voi** pensate deva funzionare, e magari si compila anche, ma non potrà mai diventare un eseguibile. Ricordate che il fatto che si compili non significa che funzioni bene.

Il nostro prossimo passo è di creare un file header per il nostro gioco. Un file header concentra le dichiarazioni di funzioni e tipi di dati in un solo posto. Questo garantisce che le strutture dati siano definite in modo consistente in modo che ogni parte del nostro programma veda ogni cosa esattamente allo stesso modo.

```
#ifndef DECK_H
#define DECK_H

#define DECKSIZE 52

typedef struct deck_t
{
  int card[DECKSIZE];
```

```
  /* numero di carte usate */
  int dealt;
}deck_t;

#endif /* DECK_H */
```

Salvate questo file come `deck.h`. Solo i file con estensione `.c` vengono compilati, quindi dovremo cambiare il nostro game.c Alla linea 2 di game.c scrivete `#include "deck.h"`. Alla linea 5 scrivete `deck_t deck;` Per avere la certezza di non aver introdotto errori, compilatelo di nuovo.

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

Nessun errore, nessun problema. Se non riuscite a compilarlo lavorateci finché non ci riuscirete.

Pre-compilazione

Come fa il compilatore a sapere che tipo è `deck_t`? Lo sa perché durante la pre-compilazione copia il file "deck.h" dentro il file "deck.c". Le direttive del pre-compilatore nel codice sorgente iniziano con "#". Potete richiamare il pre-compilatore tramite il front-end di gcc con l'opzione `-E`.

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

Quasi 3.200 linee di output! La maggior parte di queste vengono dall'include `stdio.h` ma se ci guardate dentro ci trovere anche le nostre dichiarazioni. Se non passate un nome di file con l'opzione `-o` l'output verrà mandato in console. Il processo di pre-compilazione dà maggior flessibilità al codice eseguendo tre compiti principali.

1. Copia i file "#include" nel sorgente che deve essere compilato.
2. Sistituisce i testi "#define" con il loro valore.
3. Sostituisce le macro col loro codice quando vengono chiamate.

Questo consente di avere costanti con un nome (per esempio DECKSIZE rappresenta il numero di carte in un mazzo) usate in tutto il sorgente, definite in un solo posto e aggiornate automaticamente dappertutto non appena il valore cambia. Normalmente non userete mai l'opzione `-E` da sola, ma lascerete che passi il suo output al compilatore.

Compilazione

Come passo intermedio, gcc trasforma il vostro codice in Assembly. Per farlo deve capire cosa intendevate fare analizzando il vostro codice. Se avete commesso degli errori di sintassi ve lo dirà e la compilazione fallirà. Di solito la gente confonde questo passo con l'intero processo di compilazione. Ma c'è ancora molto lavoro da fare per gcc.

Assembly

`as` trasforma il codice Assembly in codice oggetto. Il codice oggetto non può ancora essere lanciato sulla

CPU, ma ci si avvicina molto. L'opzione `-c` trasforma un file .c in un file oggetto con estensione .o Se lanciamo

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

creiamo automaticamente un file che si chiama game.o Qui siamo incappati in un punto importante. Possiamo prendere un qualsiasi file .c e creare un file oggetto da esso. Come vedremo più avanti potremo combinare questi file oggetto in un eseguibile nella fase di Link. Andiamo avanti col nostro esempio. Poiché stiamo programmando un gioco di carte abbiamo definito un mazzo di carte come un `deck_t`, creeremo una funziona per mischiare il mazzo. Questa funzione prenderà un puntatore a un tipo deck e lo riempie con delle carte messe a caso. Tiene traccia di quali carte sono già state aggiunte con l'array 'drawn'. Questo arraydi DECKSIZE elementi ci impedisce di duplicare un valore di carta.

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

static time_t seed = 0;

void shuffle(deck_t *pdeck)
{
  /* Tiene traccia di che numeri sono stati usati */
  int drawn[DECKSIZE] = {0};
  int i;

  /* Inizializzazione da fare una volta di rand */
  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);

    /* segna il valore come usato */
    drawn[value] = 1;

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

Salvate questo file come `shuffle.c`. Abbiamo inserito del codice di debug in questo codice in modo che, quando viene lanciato, scriva i valori delle carte che genera. Questo non aggiunge niente alle funzionalità del programma, ma è cruciale adesso che non possiamo vedere cosa succede. Visto che stiamo appena cominciando il nostro gioco non abbiamo altro modo di sapere se la nostra funzione fa quello che vogliamo. Con il comando printf potete vedere esattamente cosa sta succedendo in modo che quando passeremo alla prossima fase sapremo che il mazzo viene mescolato bene. Dopo che saremmo soddisfatti del buon funzionamento potremo togliere la linea dal nostro codice. Questa tecnica di debugging può sembrare

spartana, ma fa quello che deve fare con il minimo sforzo. Discuteremo di debugger più sofisticati in seguito.

Notate due cose.

1. Passiamo il parametro per indirizzo, e lo si capisce dall'operatore '&' (indirizzo di). Questo passa l'indirizzo in memoria della variabile alla funzione, in modo che la funzione possa cambiare la variabile stessa. È possibile programmare con variabili globali, ma dovrebbero essere usate molto raramente. I puntatori sono una parte importante del C e dovreste imparare a usarli molto bene.
2. Stiamo usando una chiamata a funzione da un nuovo file .c. Il Sistema Operativo cerca sempre una funzione di nome 'main' e inizia l'esecuzione da lí. `shuffle.c` non ha una funzione 'main' e quindi non può essere trasformato in un eseguibile indipendente. Dobbiamo attaccarlo ad un altro programma che abbia una 'main' e che chiami la funzione 'shuffle'.

Lanciate il comando

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

e accertatevi che crei un nuovo file chiamato `shuffle.o`. Editate il file game.c e, alla linea 7, dopo la dichiarazione della variabile `deck` di tipo deck_t, aggiungete la linea

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

Ora, se tenteremo di creare un eseguibile allo stesso modo di prima otterremo un errore

```
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 compilazione è andata a buon fine perché la sintassi era corretta. La fase di link è fallita perché non abbimo detto al compilatore dove fosse la funzione 'shuffle'. Cos'è la fase di *link* e come diciamo al compilatore dove trovare questa funzione?

Link

Il linker, `ld`, prende il codice oggetto precedentemente creato da `as` e lo trasforma in un eseguibile col comando

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

Questo unisce i due codici oggetto insieme e crea l'eeguibile `game`.

Il linker trova la funzione `shuffle` nell'oggetto shuffle.o e la include nell'eseguibile. La cosa veramente interessante dei file oggetto è il fatto che se volessimo usare di nuovo quella funzione, tutto ciò che dovremmo fare sarebbe includere il file "deck.h" e unire il file oggetto `shuffle.o` nel nuovo file eseguibile.

Il riutilizzo di codice, come in questo caso, viene fatto spesso. Non abbiamo scritto la funzione `printf` che abbiamo chiamato prima come funzione di debug, il linker ha trovato la sua definizione nel file che abbiamo incluso con `#include <stdlib.h>` e l'ha unita al codice oggetto contenuto nella libreria C (/lib/libc.so.6). In questo modo possiamo usare le funzioni di qualcun'altro, sapendo che funzionano, e preoccuparci solo di risolvere i nostri problemi. Questo è il motivo per cui i file header solitamente contengono solo le definizioni delle funzioni e non il codice vero e proprio. Normalmente si creano file

oggetto o librerie che il linker metterà nell'eseguibile. Un problema potrebbe sorgere con il nostro codice poiché non abbiamo messo nessuna definizione di funzione nel nostro header. Cosa possiamo fare per essere sicuri che tutto funzioni bene?

Altre Due Opzioni Importanti

L'opzione `-Wall` abilita tutti i tipi di avvertimenti sulla sintassi del linguaggio per essere sicuri che il nostro codice sia corretto e il più portabile possibile. Quando usiamo questa opzione e compiliamo il nostro codice vediamo qualcosa come:

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

Questo ci permette di sapere che abbiamo ancora un po' di lavoro da fare. Dobbiamo aggiungere una linea al nostro header dove diamo al compilatore tutte le informazionisulla nostra funzione `shuffle` in modo che possa fare tutti i controlli di cui ha bisogno. Sembra una difficoltà inutile, ma separa la definizione dall'implementazione e ci permette di usare la nostra funzione dove vogliamo semplicemente includendo il nostro nuovo header nel codice oggetto. Metteremo questa riga nel file deck.h

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

Questo ci libererà del messaggio di avvertimento.

Un'altra opzione comune del compilatore è l'ottimizzazione `-O#` (per esempio –O2). Questo dice al compilatore il livello di ottimizzazione che vogliamo. Il compilatore conosce un sacco di trucchi per far andare più veloce il nostro codice. Per un programma piccolo come il nostro non noterete alcuna differenza, ma per programmi più grandi potreste ottenere buoni aumenti di velocità. Lo vedete dappertutto, quindi dovreste sapere cosa significa.

Debugging

Come sappiamo tutti, il fatto che il codice venga compilato non significa che funzioni nel modo che vogliamo. Potete verificare che tutti i numeri vengano usati una sola volta lanciando

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

e controllando che non ne manchino. Cosa facciamo se c'è un problema? Come troviamo l'errore? Potete controllare il codice con un debugger. La maggior parte delle distribuzioni rendono disponibile il classico gdb. Se la linea di comando vi spaventa come lo fa a me, KDE offre una buona interfaccia con KDbg. Ci sono anche altre interfacce, e sono molto simili. Per iniziare il debugging, scegliete File–>Executable e trovate il vostro programma `game`. Quando premete F5 o selezionate Execution–>Run dal menu dovreste vedere l'output su un'altra finestra. Cosa succede? Non vediamo niente nella finestra. Non preoccupatevi, non è KDbg che non funziona. Il problema nasce dal fatto che non abbiamo messo informazioni di debug nel nostro codice eseguibile, quindi Kdbg non può dirci cosa sta succedendo internamente. L'opzione del compilatore `-g` inserisce le informazioni richieste nei file oggetto. Dovete compilare i file oggetto (estensione .o) con questa opzione, quindi il comando diventa

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

Questo inserisce degli agganci nell'eseguibile per permettere a gdb e KDbg di capire cosa sta succedendo. Il debugging è un'abilità importante, vale il tempo che dedicherete a imparare a usarla bene. Il modo in cui i debugger aiutano i programmatori è la loro capacità di mettere dei 'Breakpoint' nel codice sorgente. Provate a inserirne uno facendo click col tasto destro sulla linea che chiama la funzione `shuffle`. Un piccolo cerchio rosso dovrebbe apparire vicino alla riga. Ora, quando premete F5 il programma ferma la sua esecuzione a quella linea. Premete F8 per entrare *dentro* la funzione shuffle. Hey, ora state guardando il codice dentro `shuffle.c`! Possiamo controllare l'esecuzione passo per passo e vedere cosa sta realmente accadendo. Se passate il puntatore su una variabile locale potete vedere cosa contiene. Carino. Molto meglio di quelle istruzioni `printf`, no?

Conclusioni

Questo articolo si è presentato come un rapido giro sulla compilazione e il debugging di programmi C. Abbiamo parlato dei passi che un compilatore deve compiere e di quali opzioni passare a gcc per fargli fare tali passi. Abbiamo accennato al linking di librerie condivise e abbiamo terminato con una introduzione ai debugger. Serve ancora un sacco di lavoro per sapere esattamente cosa state facendo, ma spero che questo articolo vi sia servito per iniziare col piede giusto. Potete trovare ulteriori informazioni nelle pagine `man` e `info` di `gcc`, `as` e `ld`.

Scrivere codice da soli è la cosa migliore per imparare. Per fare pratica, potreste prendere i frammenti di codice del gioco di carte di questo articolo e scrivere un gioco di blackjack. Prendetevi il tempo di imparare come funziona un debugger. È molto più semplice partire con una interfaccia grafica come KDbg. Se aggiungerete poche funzionalità alla volta ne verrete fuori prima di quanto pensiate. Ricordate di mantenerlo funzionante!

Queste sono alcune delle cose di cui avrete bisogno per fare un gioco completo.

- Una definizione di un giocatore (per esempio, potreste definire deck_t come player_t).
- Una funzione che dia un certo numero di carte a un giocatore. Ricordate di incrementare il numero di carte date nel mazzo per tenere traccia di dove prendere la prossima carta da dare. Ricordate di terere traccia di quante carte ci sono in mano al giocatore.
- Un qualche tipo di interazione per chiedere al giocatore se vuole un'altra carta.
- Una funzione che stampi le carte del giocatore. La *carta* è valore%13 (valori da 0 a 12), il *seme* è valore/13 (valori da 0 a 3)
- Una funzione per determinare il valore della mano del giocatore. L'asso è la carta con numero 0 e può valere 1 o 11. I re hanno il numero 12 e valgono 10

Links

- gcc GCC GNU Compiler Collection
- gdb GNU Debugger
- KDbg KDE's GUI Debugger
- Award Winning Compiler Hack Ken Thompson's great compiler hack

Translation information:
en −−> −− : Lorne Bailey <sherm_pbody/at/yahoo.com>
en −−> it: Alessandro Pellizzari <alex/at/neko.it>

2005−01−10, generated by lfparser_pdf version 2.51