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

About the author:

O Lorne vive em Chicago e trabalha como consultor informático, especializado em obter dados de e para bases de dados Oracle. Desde que se mudou para um ambiente de programação *nix, evitou por completo a 'DLL Hell'. Está, presentemente a trabalhar no mestrado sobre Ciência de Computação.

GCC – a raiz para tudo

Abstract:

Este artigo assume que sabe as bases da linguagem C e introduzir−lhe−á o gcc como um compilador. Certificarnos−emos que consegue invocar o compilador a partir da linha de comandos utilizando código fonte C simples. Depois veremos muito rapidamente o que está a acontecer agora e como pode controlar a compilação dos seus programas. Daremos uma pequena entrada em como utilizar um depurador.

Regras do GCC

Consegue−se imaginar a compilar software livre com um compilador proprietário e fechado? Como é que sabe o que vai no seu executável? Pode haver qualquer tipo de "back door" ou cavalo de Tróia. O Ken Thompson, numa das piratarias de todos os tempos, escreveu um compilador que deixava uma "back door" no programa de 'login' e perpetuava o cavalo de Tróia quando o compilador se apercebia que estava a compilar a si mesmo. Leia a descrição dele para todos clássicos de todos os tempos aqui. Felizmente, temos o gcc. Sempre que faz um `configure; make; make install` o gcc faz um trabalho pesado que não se vê. Como é que fazemos o gcc trabalhar para nós? Começaremos por escrever um jogo de cartas, mas só escreveremos o necessário para demonstrar a funcionalidade do compilador. Visto que estamos a começar do zero, é preciso compreender o processo de compilação para saber o que deve ser feiro para se ter um executável e em que ordem. Daremos uma visto de olhos geral como um programa C é compilado e as opções que o gcc tem para fazer o que queremos. Os passos (e os utilitários que os fazem) são Pré−compilação (gcc −E), Compilação (gcc), Assemblagem (as), e Linkagem (ld) − Ligação.

No Princípio...

Primeiro pensamento, devíamos saber como invocar o compilador em primeiro lugar. É realmente simples. Começaremos com o clássico de todos os tempos, o primeiro programa C. (Os de velhos tempos que me perdoem).

```
#include <stdio.h>

int main()

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

Guarde este ficheiro como `game.c`. Pode compilá–lo na linha de comandos, correndo:

```
gcc game.c
```

Por omissão, o compilador C cria um executável com o nome de `a.out`. Pode corrê–lo digitando:

```
a.out
Hello World
```

Cada vez que compilar o programa, o novo `a.out` sobreporá o programa anterior. Não conseguirá dizer que programa criou o `a.out` actual. Podemos resolver este problema dizendo ao gcc o nome que queremos dar com a opção `-o`. Chamaremos este programa de `game`, contudo podíamos nomeá–lo com qualquer coisa, visto que o C não tem as restrições que o Java tem, para os nomes.

```
gcc -o game game.c

game
Hello World
```

Até este ponto, ainda estamos longe de um programa útil. Se pensa que isto é uma coisa má, deve reconsiderar o facto que temos um programa que compila e corre. À medida que adicionarmos funcionalidade, a pouco e pouco, queremos certificar–nos que o mantemos capaz de correr. Parece que todos os programadores iniciantes querem escrever 1,000 linhas de código e depois corrigi–las de uma vez só. Ninguém, mas mesmo ninguém pode fazer isto. Você faz um programa pequeno que correm faz as alterações e torna–o executável novamente. Isto limita os erros que tem de corrigir de uma só vez. E ainda por cima, você sabe exactamente o que fez e que não trabalha, então sabe onde concentrar–se. Isto evita–lhe criar algo que **você** pensa que trabalha e até compile mas nunca se torna num executável. Lembre–se que só por ter compilado não quer dizer que esteja correcto.

O nosso próximo passo é criar um ficheiro cabeçalho para o nosso jogo. Um ficheiro cabeçalho concentra os tipos de dados e a declaração de funções num só sítio. Isto assegura que as estruturas de dados estão definidas consistentemente, assim qualquer parte do nosso programa vê tudo, exactamente do mesmo modo.

```
#ifndef DECK_H
#define DECK_H

#define DECKSIZE 52

typedef struct deck_t
{
```

```
  int card[DECKSIZE];
  /* number of cards used */
  int dealt;
}deck_t;

#endif /* DECK_H */
```

Guarde este ficheiro como `deck.h`. Só o ficheiro `.c` é que é compilado, assim temos de alterar o nosso game.c. Na linha 2 do game.c, escreva `#include "deck.h"`. Na linha 5, escreva `deck_t deck;` para ter a certeza que não falhámos nada, compile−o novamente.

```
gcc −o game game.c
```

Se não houver erros, não há problema. Se não compilar resolva−o até compilar.

Pré−compilação

Como é que o compilador sabe que tipo é o `deck_t` ? Porque durante a pré−compilação, ele, na verdade copia o ficheiro "deck.h" para dentro do ficheiro "game.c". As directivas do pré−compilador no código fonte começam por um "#". Pode invocar o pré−compilador através do frontend do gcc com a opção `−E`.

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

Praticamente 3,200 linhas de saída! A maioria delas vem do ficheiro incluído `stdio.h`, mas se der uma vista de olhos nele, as suas declarações também lá estão. Se não der um nome de ficheiro para saída com a opção `−o`, ele escreve para a consola. O processo de pré−compilação dá mais flexibilidade ao código ao atingir três grandes objectivos.

1. Copia os ficheiros "#included" no código fonte para serem compilados.
2. Substirui o texto "#define" pelo seu valor actual.
3. Substitui as macros nas linhas onde são chamadas.

Isto permite−lhe ter constantes com nomes (por exemplo, a DECKSIZE representa o número de cartas num baralho) utilizadas ao longo do código e definidas num só sítio e automaticamente actualizadas sempre que o seu valor se altera. Na prática, quase num utilizará a opção `−E` isoladamente, mas deixará passar a sua saída para o compilador.

Compilação

Como um passo intermediário, o gcc traduz o seu código em linguagem Assembler. Para fazer isto, ele deve descobrir o que é que tinha intenção de fazer ao passar por todo o seu código. Se cometer um erro de sintaxe ele dir−lhe−à e a compilação falhará. Muitas vezes as pessoas confundem este passo com todo o processo inteiro. Mas ainda há mais trabalho para o gcc fazer.

Assemblagem

O `as` transforma o código Assembler em código objecto. O código objecto não pode ainda correr no CPU, as está muito perto. A opção do compilador `-c` transforma um ficheiro .c num ficheiro objecto com a extensão .o. Se correrrmos:

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

criamos automaticamente um ficheiro chamado game.o. Aqui caímos num ponto importante. Podemos tomar qualquer ficheiro .c e criar um ficheiro objecto a partir dele. Como vemos abaixo, podemos combinar estes ficheiros objecto em executáveis no passo de Ligação. Continuemos com o nosso exemplo. Visto estarmos a programar um jogo de cartas e definimos um baralho de cartas como um `deck_t`, escreveremos um função para baralhar o baralho. Esta função recebe um ponteiro do tipo deck e correga–o com um valores aleatórios para as cartas. Mantém rasto das cartas já desenhadas, com o vector de 'desenho'. Este vector com membros do DECKSIZE evita–nos duplicar o valor de uma carta.

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

static time_t seed = 0;

void shuffle(deck_t *pdeck)
{
  /* Keeps track of what numbers have been used */
  int drawn[DECKSIZE] = {0};
  int i;

  /* One time initialization of 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);

    /* mark value as used */
    drawn[value] = 1;

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

Guarde este ficheiro como `shuffle.c`. Pusemos uma frase de depuração no código para quando correr, escrever o número das cartas que gera. Isto não adiciona nenhuma funcionalidade ao programa, mas agora, é crucial para vermos o que se passa. Visto estarmos somente a começar o nosso jogo, não temos outro modo senão que termos a certeza que a nossa função está a fazer o que pretendemos. Com a frase printf, podemos

ver exactamente o que está acontecer e assim quando passarmos para a próxima fase sabemos que o baralho esta bem baralhado. Depois de estarmos satisfeitos com o seu funcionamento podemos remover esta linha do nosso código. Esta técnica de fazer depuração aos programas parece arcaica mas fá−lo com um mínimo de trabalho irrelevante. Discutiremos mais tardes depuradores mais sofisticados.

Note duas coisas.

1. Passamos por parâmetro o seu endereço, pode dizer isto pelo '&' (endereço de) operador. Isto passa o endereço de máquina da variável para a função, assim a função pode alterar a própria variável. É possível programar com variáveis globais, mas deviam ser utilizadas raramente. Os ponteiros são uma parte importante do C e devia entende−los também.
2. Estamos a utilizar uma chamada de função a partir do novo ficheiro .c. O sistema operativo procura sempre por uma função chamada 'main' e começa a sua execução aí. O `shuffle.c` não tem uma função 'main' por conseguinte não pode ser um executável por si só. Devemos combiná−lo com outro programa que tenha uma função 'main' e que chame a função shuffle.

Corra o comando

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

e certifique−se que cria um novo ficheiro chamado `shuffle.o`. Edite o ficheiro game.c, e na linha 7, após a declaração da variável deck_t `deck`, adicione a linha

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

Agora, se tentarmos criar um executável do mesmo modo como antes obtemos um erro

```
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
```

O compilador teve sucesso porque a nossa sintaxe estava correcta. A fase de ligação falhou porque não dissemos ao compilador onde se encontra a função 'shuffle'. O que é que é a *ligação* e como é que dizemos ao compilador onde pode encontrar esta função? Ligação

O linker, `ld`, pega no código objecto previamente criado com `as` e transforma−o num executável através do comando

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

Isto combinará os dois objectos e criará o executável `game`.

O linker encontra a função `shuffle` a partir do objecto shuffle.o e inclui−o no executável. A verdadeira beleza dos ficheiros objecto vem do facto se quisermos utilizarmos esta função novamente, só temos de incluir o ficheiro "deck.h" e ligar ao código do novo executável o ficheiro objecto `shuffle.o`.

O aproveitamento do código está sempre a acontecer. Não escrevemos o código da função `printf` quando a chamámos em cima como uma declaração de depuração, O linker encontra a sua definição no ficheiro que incluímos `#include <stdlib.h>` e liga−o ao código objecto armazenada na biblioteca C (/lib/libc.so.6). Deste modo podemos utilizar a função de alguém que sabemos trabalhar correctamente e preocuparmo−nos em resolver os nossos problemas. É por este motivo que os ficheiros de cabeçalho só contêm as definições de dados e de funções e não o corpo das funções. Normalmente você cria os ficheiros objecto ou bibliotecas para

o linker por no executável. Um problema podia ocorrer com o nosso código visto que não pusemos nenhum definição no nosso ficheiro cabeçalho. O que é que podemos fazer para ter a certeza que tudo corre sem problemas?

Mais duas Opções Importantes

A opção `-Wall` activa todo o tipo de avisos relativamente à sintaxe da linguagem para nos ajudar a ter a certeza que o nosso código está correcto e portável tanto quanto possível. Quando utilizamos esta opção e compilamos o nosso código podemos ver algo como:

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

Isto diz–nos que temos mais algum trabalho a fazer. Precisamos de pôr uma linha no ficheiro de cabeçalho, onde diremos ao compilador tudo sobre a nossa função `shuffle` assim poderá fazer as verificações que precisa de fazer. Parece complicado, mas separa a definição da implementação e permite–nos utilizar a função em qualquer lado bastando incluir o nosso novo ficheiro cabeçalho e ligá–lo ao nosso código objecto. Introduziremos esta linha no ficheiro deck.h.

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

Isto evitará a mensagem de aviso.

Uma outra opção comum do compilador é a optimização `-O#` (ou seja –O2). Isto diz ao compilador o nível de optimização que quer. O compilador tem um saco cheio de truques para tornar o seu código mais rápido. Para um pequeno programa como o nosso não notaremos qualquer diferença, mas para programas grandes pode melhorar um pouco a rapidez. Você vê esta opção em todo o lado por isso devia saber o que significa.

Depuração

Como todos sabemos, só porque o nosso código compilou não quer dizer que vai trabalhar do modo que queremos. Pode verificar que são utilizados todos os números de uma só vez correndo

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

e vendo que não falta nada. O que é que devemos fazer se houver um problema? Como é que olhamos por debaixo da madeira e encontramos o erro? Pode verificar o seu código com um depurador. A maioria das distribuições fornecem um depurador clássico, o gdb. Se a linha de comandos o atrapalha como a mim, o KDE oferece um front–end bastante simpático, com o KDbg. Existem outros front–ends, e são muito semelhantes. Para começar a depurar, escolha File–>Executable e depois encontre o seu programa, o `jogo`. Quando prime F5 ou escolhe Execution–>Run a partir do menu, você devia ver uma saída numa janela à parte. O que é que acontece ? Não vemos nada na janela. Não se preocupe, o KDbg não está a falhar. O problema vem do facto de não termos posto nenhuma informação de depuração no executável, assim i KDbg não nos pode dizer o que se passa internamente. A flag do compilador `-g` põe a informação necessária dentro dos ficheiros objecto. Deve compilar os ficheiros objecto (extensão .o) com esta flag, assim o comando passa a ser:

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

Isto põe marcas no executável que permitem ao gdb e ao KDbg saber o que está a fazer. Fazer depuração é uma tarefa importante e vale a pena o tempo gasto em aprender como o fazer correctamente. O modo como os depuradores ajudam os programadores é a habilidade de definir "pontos de paragem" no código fonte. Tente agora definir um através de um clique com o botão esquerdo na linha com a chamada à função `shuffle`. Deve aparecer um pequeno circulo vermelho na linha seguinte. Agora, prime F5 e o programa pára a execução nessa linha. Prima F8 para entrar *dentro* da função shuffle. Bem, estamos agora a olhar para o código a partir do ficheiro `shuffle.c`! Podemos controlar a execução passo a passo e ver o que se passa. Se deixar o apontador sobre uma variável local, verá o valor que guarda. Apreciável. Muito melhor que aquelas frases com `printf's`, não é?

Resumo

Esta artigo apresentou uma visita clara à compilação e depuração aos programas C. Discutimos os passos que o compilador faz e as opções que devemos passar ao gcc para ele fazer esses passos. Introduzimos a ligação a bibliotecas partilhadas e terminámos com uma introdução aos depuradores. Requer bastante trabalho para saber realmente, o que está a fazer, mas espero que isto o ajude a começar com o pé direito. Pode encontrar mais informação nas páginas `man` e `info` acerca do `gcc`, `as` e do `ld`.

Escrever código por si mesmo ensina−lhe o mais importante. Para praticar, podia utilizar estas bases simples do programa de jogo de cartas utilizado neste artigo e escrever um jogo de blackjack. Aproveite o tempo para aprender a utilizar um depurador. É muito mais fácil começar com um GUI como o KDbg. Se adicionar funcionalidade aos poucos, saberá as coisas sem se dar por isso. Lembre−se mantenha−o a correr!

Aqui estão algumas coisas que poderá precisas para criar um jogo completo.

- Uma definição do jogador de cartas (por exemplo podia definir o deck_t como definiu o player_t).
- Uma função que distribui um dado número de cartas a um dado jogador. Lembre−se de incrementar o número de 'distribuídas' do baralho para saber de onde deve tirar a próxima carta. Lembre−se de saber o número de cartas que cada jogador tem na mão.
- Alguma interacção dos utilizadores, para pedir se o utilizador quer outra carta.
- Uma função para apresentar as cartas de um jogador. A *carta* tem um valor de % 13 (a começar em 0 até 12), o *naipe* tem um valor de / 13 (a começar em 0 até 3).
- Uma função para determinar o valor das cartas que um utilizador tem na mão. Os ases são cartas com um valor de zero e podem valer 1 ou 11. Os reis têm valor 10 e no mínimo 10.

Ligações

- gcc Colecção do compilador GCC da GNU
- gdb O Debugger da GNU
- KDbg Debugger GUI do KDE
- Compilador pirata Grande compilador pirata de Ken Thompson