

# GCC - der Ursprung von allem



Version vom 01.07.2003 - Adam Diez (diez@gsmn.de)

## *Zusammenfassung:*

Dieser Artikel geht davon aus, dass Du die Grundlagen der Programmiersprache C beherrscht, und wird Dir zeigen, wie Du den gcc als Compiler verwendest. Wir werden dafür sorgen, dass Du den Compiler für einfachen C-Code direkt von der Kommandozeile aufrufen kannst. Danach werden wir einen Blick darauf werfen, was gerade passiert, und wie Du die Kompilierung Deiner Programme kontrollieren kannst. Außerdem werden wir einen Blick auf die Verwendung eines Debuggers werfen. Dieses Dokument richtet sich an diejenigen die nun ihre ersten Schritte in der Programmiersprache C beginnen.

---

## GCC ist klasse!

Kannst Du Dir vorstellen, Freie Software mit einem kommerziellen, "closed Source" Compiler zu kompilieren? Woher weist Du, was in deinem ausführbarem Programm passiert? Es könnte eine Art Hintertür oder ein Trojaner eingebaut werden. Ken Thompson, schrieb für einen der größten Hacks aller Zeiten, einen Compiler, der eine Hintertür in das 'login' programm einbaute und den Trojaner verewigte, als der Compiler merkte, dass er sich selbst kompilierte. Lies seine Beschreibung dieses Klassikers [1]. Glücklicherweise haben wir gcc. Immer wenn Du ein `configure; make; make install` ausführst, macht gcc eine Menge Arbeit im Hintergrund. Wie lassen wir gcc für uns arbeiten? Wir werden damit anfangen, ein Kartenspiel zu programmieren, aber wir werden nur soviel schreiben, um die Funktionalität des Compilers zu demonstrieren. Da wir von Grund auf starten, haben wir gute Voraussetzungen dazu, den Kompilierungsprozess zu verstehen und was in welcher Reihenfolge passiert, um ein ausführbares Programm zu erzeugen. Wir werden einen Überblick bekommen, wie ein C-Programm kompiliert wird, und die Optionen, um dem gcc zu sagen, was wir wollen. Die Schritte (und die Werkzeuge, die das machen ) sind [Vor-Kompilieren](#) (gcc -E), [Kompilieren](#) (gcc), [Assemblieren](#) (as), und [Linken](#) (ld).

## Zum Anfang ...

Zuerst sollten wir wissen wie man den Compiler aufruft. Es ist wirklich einfach. Wir werden mit dem klassischen ersten C-Programm anfangen. (Fortgeschrittene müssen mir vergeben).

```
#include <stdio.h>

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

Speichere diese Datei als `game.c`. Du kannst sie in der Kommandozeile übersetzen mit:

```
gcc game.c
```

Defaultmäßig generiert der C Compiler eine ausführbare Datei namens `a.out`. Ausführen kannst Du es mit:

```
a.out
Hello World
```

Jedes Mal, wenn Du ein Programm kompilierst, überschreibt das neue `a.out` das vorige Programm. Du wirst nicht wissen, von welchem Programm das derzeitige `a.out` ist. Wir können dieses Problem lösen, indem wir gcc mit der Option `-o` mitteilen, wie das erzeugte Programm heißen soll. Wir werden das Programm `game` nennen, obwohl wir es irgendwie nennen könnten, da C keine Einschränkungen in der Wahl des Namens hat wie etwa Java.

```
gcc -o game game.c
game
Hello World
```

An diesem Punkt sind wir immer noch ein gutes Stück davon entfernt, ein sinnvolles Programm zu haben. Wenn Du jetzt denkst, dass sei schlecht, solltest Du berücksichtigen, dass wir ein Programm haben, das sich kompilieren lässt und läuft. Wenn wir jetzt Stück für Stück Funktionalität in das Programm bringen, dann nur um sicher zu gehen, dass es ausführbar bleibt. Es scheint so zu sein, dass jeder angehende Programmierer zuerst 1000 Zeilen Sourcecode schreiben, und dann alles auf einmal korrigieren will. Niemand, und ich meine Niemand kann das. Du schreibst ein kleines Programm, das funktioniert, du änderst es und lässt es wieder laufen. Das schränkt die Anzahl der Fehler, die Du auf einmal korrigieren mußt, ein. Und, Du weißt genau, was Du gerade getan hast, damit Du es korrigieren kannst. Dies hält Dich davon ab, etwas zu erzeugen, von dem **Du** denkst, dass es arbeiten sollte, und auch kompiliert werden kann, aber niemals laufen wird. Erinnerung Dich, nur weil es kompiliert wird, ist es nicht richtig.

Unser nächster Schritt ist, ein Header-File für unser Spiel zu erzeugen. Ein Header-File sammelt Datentypen und Funktionsdeklarationen an einem Ort. Dies stellt sicher, dass die Definitionen der Datenstrukturen übereinstimmen, so dass jeder Teil unseres Programms genau das gleiche sieht.

```
#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 */
```

Speichere diese Datei als `deck.h`. Nur `.c` Dateien werden kompiliert, also müssen wir unsere Datei `game.c` ändern. In Zeile 2 der Datei `game.c`, schreib `#include "deck.h"`. In Zeile 5 schreib `deck_t deck`; Um sicher zu gehen, dass wir nichts zerstört haben, kompilieren wir es nochmal.

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

Keine Fehler, kein Problem. Wenn es nicht kompilierbar ist, arbeite daran, bis es funktioniert.

## Vor-Kompilieren

Wie weiß der Compiler, was ein `deck_t` Typ ist? Während der Vor-kompilierung, wird die Datei `"deck.h"` in die Datei `"game.c"` kopiert. Die Vor-kompilierer Anweisungen sind durch ein `"#"` gekennzeichnet. Du kannst den Precompiler (Die Vor-kompilierer) über den gcc mit der Option `-E` aufrufen.

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

3,200 Zeilen Ausgabe! Das meiste davon kommt von der `stdio.h` include-Datei, aber wenn Du es Dir näher ansiehst, sind unsere Deklarationen auch darin. Wenn Du keinen Namen für die erzeugte Datei mittels `-o` Option angibst, kommt die Ausgabe ins Textfenster. Der Vorgang der Vorkompilierung gibt dem Code eine größere Flexibilität durch Erreichung folgender Ziele:

- Kopieren der `"#include"` Dateien in das zu kompilierende Source-File.
- Ersetzen der `"#define"` Texte mit den aktuellen Werten.

- Ersetzen der Macros in der Zeile wann immer Sie aufgerufen werden. Dies erlaubt Dir, *Konstanten* zu verwenden (z.B.: DECKSIZE entspricht der Anzahl an Karten in einem Blatt), die im ganzen Code verstreut sind, an einer Stelle zu deklarieren, und automatisch zu übernehmen, immer wenn Du den Wert änderst. In der Praxis wirst Du nie die -E Option direkt verwenden, jedoch wirst Du den Output dem Compiler zukommen lassen.

## Kompilieren

Als Zwischenschritt übersetzt gcc deinen Code in Assembler-Code. Um das zu tun, muss er ausrechnen, was Du tun wolltest, indem es deinen Code übersetzt. Wenn Du einen Syntax-Error machst, wird er es Dir das mitteilen und der Kompilierungsvorgang wird abgebrochen. Manche Leute glauben, dass dies der einzige Schritt im Kompilierungsvorgang ist, aber es gibt noch mehr für gcc zu tun.

## Assemblieren

as übersetzt den Assembler Code in Object-Code. Object-Code kann nicht am Prozessor verarbeitet werden, aber er ist schön geschlossen. Die Compiler Option -c verwandelt eine .c Datei in ein Object-File mit einer .o Endung. Wenn wir

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

aufrufen, erzeugen wir automatisch eine Datei namens game.o. Hier sind wir an einem wichtigen Punkt angelangt. Wir können jede .c Datei nehmen und eine Object-Datei daraus erzeugen. Wie wir unten sehen, können wir diese Object-Files im Linker-Vorgang zu einem ausführbaren Programm machen. Lass uns mit unserem Beispiel weitermachen. Da wir ein Kartenspiel programmieren und Kartenspiel definiert haben mit deck\_t, werden wir eine Funktion schreiben, die unser Kartenspiel mischt. Diese Funktion legt den Zeiger auf eine Kartenart und lädt ihn mit Zufallswerten für die Kartenwerte. Sie merkt sich auch, welche Karten bereits verwendet wurden mit dem 'drawn' array. Dieses array mit DECKSIZE Mitgliedern, bewahrt uns davor, einen Kartenwert doppelt zu verwenden.

```
#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;
}

```

Speichere diese Datei als `shuffle.c`. Wir haben eine Debug-Anweisung in den Code eingefügt, damit gibt es uns, wenn das Programm läuft, die Kartenwerte aus, die es generiert. Dies verändert zwar nicht die Funktionalität unseres Programms, aber es ist entscheidend, da wir wissen, was jetzt geschieht. Da wir gerade erst mit dem Spiel anfangen, haben wir keine andere Möglichkeit, um zu testen, ob unser Programm das tut, was wir wollen. Mit der `printf` Anweisung können wir exakt sehen, was gerade passiert, also können wir jetzt mit der nächsten Phase beginnen, wo wir wissen, dass unser Blatt gut gemischt ist. Nachdem wir uns daran erfreut haben, dass es richtig arbeitet, können wir diese Zeile wieder aus dem Code entfernen. Die Art, ein Programm zu debuggen, wirkt zwar etwa ordinär, aber es ist die unkomplizierteste Art. Wir werden aufregendere debugger später kennenlernen.

Notiere Dir zwei Dinge:

1. Wir schreiben eine Variable mit ihrer Adresse, welche man mit dem '&' (Adresse von) Operator erhält. Diese übergibt die Maschinenadresse der Variablen an die Funktion, also kann die Funktion selbst die Variable ändern. Es ist zwar möglich mit globalen Variablen zu arbeiten, aber diese sollten nur selten verwendet werden. Zeiger sind ein wichtiger Teil von C und sollten gut verstanden werden.
2. Wir verwenden einen Funktionsaufruf von einer neuen `.c` Datei. Das Betriebssystem schaut immer nach einer Funktion namens "main" und startet dort. `shuffle.c` hat keine Funktion "main", und vorher kann kein ausführbares Programm erzeugt werden. Wir müssen es mit einem anderen Programm kombinieren, das eine Funktion "main" hat, und die Funktion "shuffle" aufruft.

Führe folgenden Befehl aus:

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

und überzeuge Dich, dass eine neue Datei namens `shuffle.o` erzeugt wird. Editiere die `game.c` Datei, und in Zeile 7, nach der Deklaration der `deck_t` variable `deck`, füge folgende Zeile ein:

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

Wenn wir jetzt versuchen, eine ausführbare Datei zu erzeugen, bekommen wir eine Fehlermeldung:

```

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

```

Das Kompilieren funktionierte, da unsere Syntax richtig war. Das Linken schlug fehl, weil wir dem Compiler nicht gesagt haben wo die 'shuffle' Funktion ist. Was ist der *link* und wie sagen wir dem Compiler, wo er die Funktion finden kann?

## Linken

Der Linker, `ld`, nimmt den Object-Code, welcher zuvor von `as` erzeugt wurde, und wandelt es in ein ausführbares Programm um mit dem Befehl

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

Dies wird die zwei Objekte zusammenführen und die ausführbare Datei `game` erzeugen. Der Linker findet die `shuffle` Funktion vom `shuffle.o` Objekt und fügt es in die ausführbare Datei ein. Das wirklich tolle an den Object-Files ist, dass, wenn wir die Funktion wieder verwenden wollen, wir nur noch die "`deck.h`" Datei einfügen und das `shuffle.o` Object-File zur neuen auszuführenden Datei dazulinken müssen. Wiederverwendung von Code auf diese Art ist durchaus üblich. Wir haben nicht die `printf` Funktion geschrieben, die wir oben zum Debuggen verwendet haben, aber der Linker findet die Definitionen in dem File, das wir mit `#include <stdlib.h>` eingebunden haben, und zeigt zu dem Object-Code in der C-Bibliothek

(`/lib/libc.so.6`). Auf diese Art können wir Funktionen von jemand anderem verwenden, ohne uns Sorgen zu machen, ob sie funktionieren, und uns um unsere eigenen Probleme kümmern. Dies ist der Grund, weshalb Header-Dateien normalerweise die Daten- und Funktionsdeklarationen, aber nicht die Funktionen selbst enthalten. Normalerweise machst Du Object-Files bzw. Bibliotheken für den Linker, um es ins Programm zu schreiben. Ein Problem könnte auftreten, da wir nicht alle Funktionsdefinitionen in unseren Header geschrieben haben. Was können wir tun, um zu überprüfen, ob alles in Ordnung ist?

## Zwei weitere wichtige Optionen

Die `-Wall` Option schaltet alle Warnungen betreffend Syntax ein, damit wir sicher sein können, dass unser Code in Ordnung und so weit wie möglich portierbar ist. Wenn wir diese Option verwenden und unseren Code kompilieren, sehen wir etwas ähnliches wie:

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

Dies teilt uns mit, dass wir ein wenig mehr zu tun haben. Wir müssen eine Zeile in ein Header-File einfügen, in der wir dem Compiler alles über unsere `shuffle` Funktion mitteilen, damit der Compiler alles überprüfen kann, was er überprüfen soll. Es klingt lästig, aber es trennt die Definition von der Implementation und erlaubt uns, unsere Funktion überall anders zu verwenden, indem wir einfach einen neuen Header einbinden, und zu unserem Object Code hinzulinken. Wir schreiben diese eine Zeile in unsere `deck.h` Datei.

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

Dies beseitigt die Warn-Ausgaben.

Eine weitere Compiler-Option ist die Optimierung. `-O#` (z.B.: `-O2`). Dies teilt dem Compiler mit, welche Stufe der Optimierung Du möchtest. Der Compiler hat eine Menge Tricks, um Deinen Code ein bißchen schneller zu machen. Bei kleinen Programmen wie unserem hier wirst Du keine Unterschiede merken, größere Programme jedoch werden etwas kleiner. Du wirst es überall sehen, deshalb solltest Du auch wissen, was es bedeutet.

## Debuggen

Wie wir alle wissen, heißt es nicht, wenn unser Code kompiliert wird, dass er auch so arbeitet, wie wir wollen. Du kannst überprüfen, ob alle Nummern verwendet werden, wenn du folgendes ausführst

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

und überprüfst, dass nichts fehlt. Was tun wir, wenn etwas fehlt? Was tun wir, wenn ein Problem auftritt? Wie können wir etwaige Fehler finden?

Du kannst Dein Programm mit einem Debugger überprüfen. Die meisten Distributionen verwenden den klassischen Debugger `gdb`. Wenn Dir die Anzahl der Optionen auf der Kommandozeile zu viel sind, bietet KDE eine sehr schöne grafische Oberfläche für den `gdb` an namens `KDbg` [2]. Es gibt auch andere grafische Oberflächen, und sie sind sich alle sehr ähnlich. Um mit dem Debugging zu beginnen, wählst Du `File->Executable` und suchst dann Dein `game` Programm. Wenn Du `F5` drückst, oder `Execution->Run from the menu`, solltest Du eine Ausgabe in einem anderen Fenster sehen. Was ist passiert? Wir sahen nichts im Fenster. Sorg Dich nicht, `KDbg` ist nicht kaputt. Das Problem kommt daher, dass wir keinerlei Debugg-Information in unserem Programm haben, also kann uns `KDbg` nicht sagen, was gerade passiert. Das Compiler Flag `-g` gibt die notwendigen Ausgaben in das Object-File. Du must das Object-File (`.o` Endung) mit diesem Flag kompilieren, also lautet der Befehl

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

Dies markiert Sachen im ausführbaren Programm so, dass `gdb` und `KDbg` in der Lage sind, anzuzeigen, was gerade passiert. Debuggen ist eine wichtige Fähigkeit, es ist sicher wichtig zu lernen einen zu benutzen. Debugger helfen dem Programmierer mit der Möglichkeit "Breakpoints" im Source Code zu setzen. Versuche nun einen zu setzen, indem Du mit der rechten Maustaste auf die Zeile klickst, die die `shuffle` Funktion aufruft. Ein kleiner roter Kreis sollte neben der Zeile erscheinen. Wenn Du jetzt `F5` drückst, bleibt das Programm an dieser Stelle stehen. Drücke `F8` um *in* die `shuffle` Funktion zu kommen. Hey, jetzt siehst Du den Code von `shuffle.c`! Wir können

die Abarbeitung des Programms Schritt für Schritt kontrollieren und schauen was wirklich passiert. Wenn Du den Mauszeiger über eine lokale Variable stellst, siehst Du, was sie beinhaltet. Süß. Es ist viel besser als diese `printf` Anweisungen, oder?

## Zusammenfassung

Dieser Artikel war eine rasante Tour durch Kompilieren und Debuggen von C- Programmen. Wir diskutierten die Schritte, die der Compiler durchläuft und welche Optionen gcc verwendet, um diese Schritte zu machen. Wir streiften das Linken mit shared libraries und endeten mit einer Einführung über Debugger. Es nimmt viel Zeit in Anspruch, zu lernen, was Du tust, doch ich hoffe, dies hat dir geholfen, richtig zu beginnen. Mehr Informationen zu dem Thema findest Du im man und in den info Seiten für gcc, as und ld.

Selbst zu programmieren, lehrt am meisten. Zur Übung kannst Du die einfachen Anregungen für das Kartenspielprogramm in diesem Artikel verwenden und ein Black Jack Spiel programmieren. Nimm Dir die Zeit, um zu lernen, wie man den Debugger verwendet. Es ist viel einfacher mit einem GUI wie KDbg anzufangen. Wenn Du ein bisschen Funktionalität auf einmal einbringst, wirst Du fertig sein, bevor Du es merkst. Denk daran, halt es lauffähig!

Hier einige Dinge, die Du brauchen wirst, um ein vollständiges Spiel zu programmieren:

3. Eine Definition eines Spielers (Du könntest z.B.: `deck_t` als einen `player_t` definieren).
4. Eine Funktion, die eine bestimmte Anzahl von Karten an einen bestimmten Spieler austeilt. Denk daran, die Zahl der 'ausgespielten' im Blatt zu merken, damit Du weißt, von wo Du die nächste Karte nehmen sollst. Denk daran Dir zu merken, wie viele Karten ein Spieler in der Hand hat.
5. Eine Abfrage, ob der Spieler eine weitere Karte möchte.
6. Eine Funktion, um die Karten eines Spielers darzustellen. Der `card`-Wert ist % 13 (von 0 bis 12), der `suit`-Wert ist Wert / 13 (von 0 bis 3).
7. Eine Funktion, um den Wert eines Blattes zu bestimmen. Assen haben die Nummer Null und können 1 oder 11 wert sein sein. Könige haben die Nummer 12 und den Wert 10.

## Linkverzeichnis

- [1] - <http://www.acm.org/classics/sep95>
- [2] - <http://members.nextra.at/johsxt/kdbg.html>

## Schlusswort

Ich bitte darum Änderungsvorschläge, Erweiterungen, Interesse an Mitarbeit und Anmerkungen an die eMail-Adresse [diez@gsmn.de](mailto:diez@gsmn.de) zu senden. Das Projekt <http://www.unix-root.de> stellt die aktuellste Version dieses Dokuments, in verschiedenen Dateiformaten zum Download bereit. Des weiteren, bedankt sich der Autor beim Online Linuxmagazin „[Linux-Focus](#)“.

Das Originaldokument wurde von Lorne Bailey erstellt, von Rene Süß ins deutsche übersetzt und nun von Adam Diez weiterentwickelt und verwaltet. Es steht unter der GPL.