14
Fehler finden, Fehler vermeiden

Jörn Loviscach

Versionsstand: 29. September 2012, 20:53

Die nummerierten Felder sind absichtlich leer, zum Ausfüllen beim Ansehen der Videos:
http://www.j3L7h.de/videos.html

1 Turings Halteproblem

Es ist schwer, Programme automatisch auf Korrektheit zu prüfen. Eine grundlegende Erkenntnis dazu stammt von Alan Turing (1912–1954). Um diese zu verstehen, kann man eine simple Art Programme betrachten: Programme, die jede beliebig, aber endlich lange Datei einlesen, die verarbeiten und dann eine Zahl ausgeben – falls sie nicht endlos laufen. Hier zwei Beispiele für solche Programme:

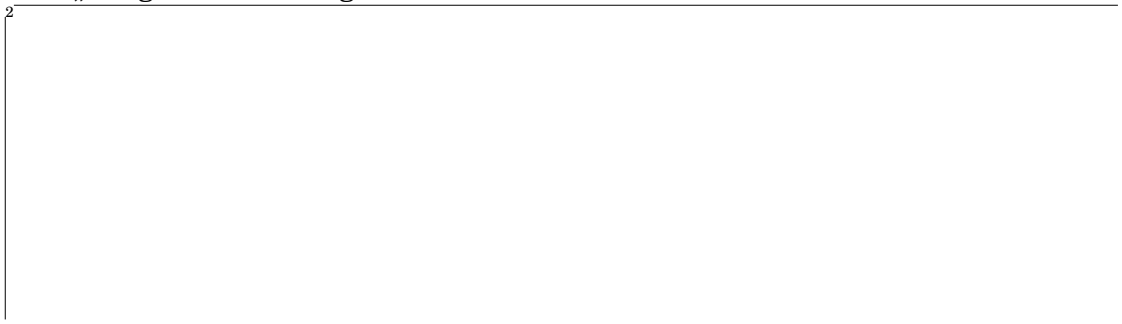
1

Zur weiteren Vereinfachung nimmt man an, dass der Rechner immer genug Speicher hat. Zum Beispiel kann man sich das so vorstellen: Hat ein Programm zu wenig Speicher, hält es an und wartet, bis man genug Speicher nachgerüstet hat. Der Rechner hat dann effektiv beliebig viel Speicher – allerdings zu jedem Zeitpunkt immer nur endlich viel.

Auch Programme sind als Dateien gespeichert. Man kann also jedes dieser Programme auch mit dem Code eines anderen füttern – oder sogar mit seinem eigenen Code. Insbesondere könnte es vielleicht (wohlgemerkt: vielleicht!) ein Programm P geben, das einem sagt, ob ein beliebiges gegebenes Programm angewendet auf eine gegebene Eingabedatei in endlicher Zeit endet (= hält) oder

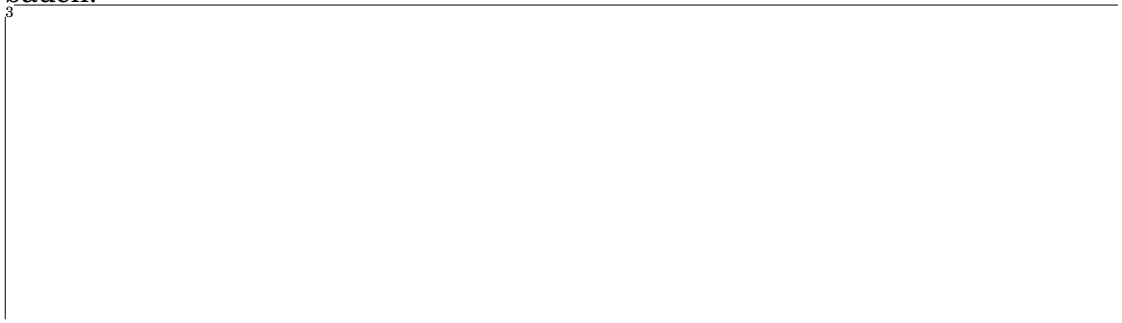
aber „hängt“. Dieses Programm P könnte etwa so zu bedienen sein:

[2]



Wenn es ein solches Programm P gäbe, dann könnte so ein anderes Programm Q bauen:

[3]

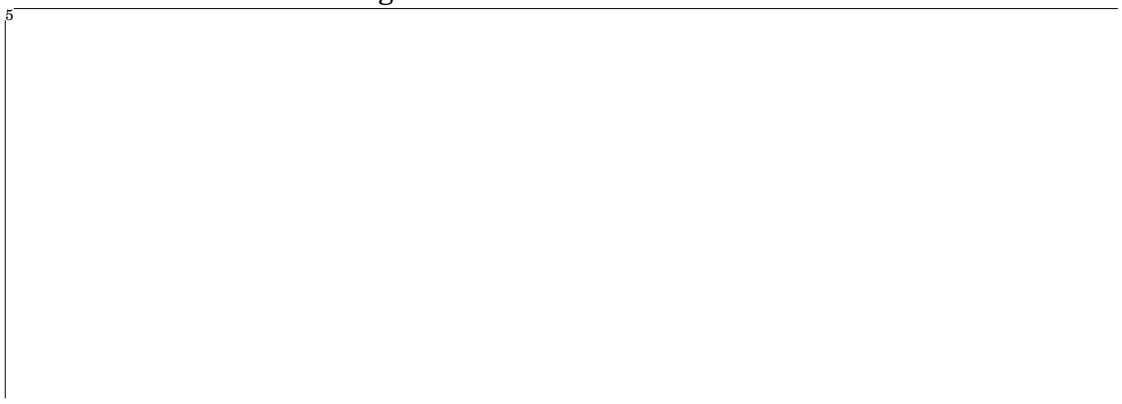


Nun kann man sich fragen, ob das Programm Q endet, wenn man es mit seiner eigenen Datei Q füttert.

[4]

Dieses Resultat ist allerdings etwas theoretisch:

[5]



2 Code Conventions

Ein wesentlicher Teil der Fehlervermeidung besteht darin, Programmcode so zu schreiben, dass man ihn leicht lesen kann – und leicht bearbeiten („warten“) kann. Gerade bei Team-Projekten ist dazu sinnvoll, sich auf einen Stil zu einigen. Code Conventions, Coding Standards oder Style Guides enthalten dazu rein formale Vorschriften wie vielleicht diese:

[6]

Daneben gibt es darin Vorschriften, die fehlerträchtige Konstruktionen vermeiden sollen:

[7]

Beispiele: GNU Coding Standards (Link) und Googles C++ Style Guide (Link) In der Windows-Programmierung war lange Zeit die „ungarische Notation“ üblich, bei der man jedem Variablennamen ein Kürzel für den Typ voranstellt: `lpszMenuName`, `dwStyle`. Einige solcher Vorschriften lassen sich auch maschinell prüfen. Die IAR Embedded Toolbench kann das für die Vorschriften von MISRA (Motor Industry Software Reliability Association), ein anfangs britischer Verband. Die entsprechenden Optionen unter Options > General Options > MISRA-C lassen sich aber nicht in der Gratisversion der IAR Embedded Toolbench nutzen.

Man kann auch einen Sport daraus machen, unlesbaren C-Code zu schreiben. Beim International Obfuscated C Code Contest (Link) findet sich zum Beispiel ein Programm, das nur scheinbar (!) Primzahlen berechnet (Link), und ein Programm, das ein Briefwechsel zwischen `char*lie` und `(char)lotte` ist (Link).

3 Defensive Programmierung

Ein wenig Umsicht beim Programmieren erspart Nächte bei der Fehlersuche. Zuallererst gehört dazu, die Möglichkeiten des Compilers auszureizen:

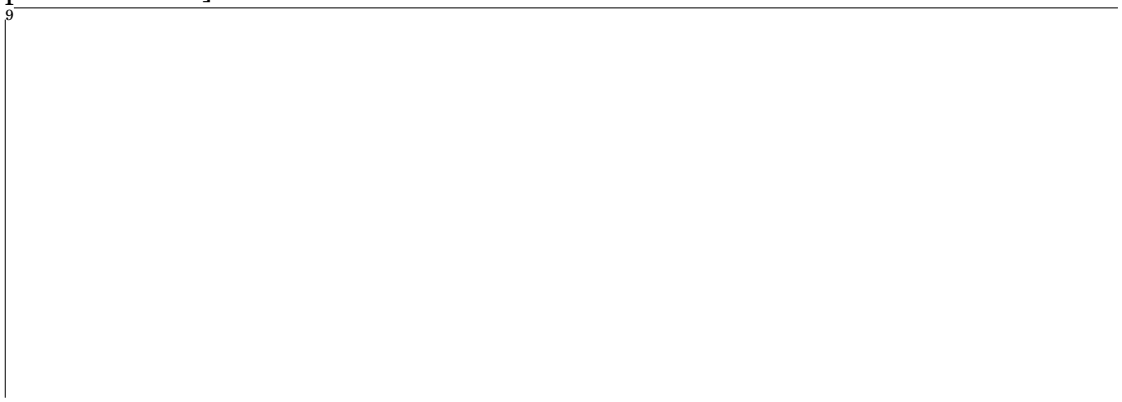
8

Der nächste Schritt ist, dafür zu sorgen, dass das Programm frühestmöglich selbst merkt, dass etwas schief läuft. Es soll nicht zum Beispiel falsche Daten in eine Datenbank schreiben und erst fünf Minuten später durch einen Folgefehler abstürzen. Dann sind nicht nur die Daten zerstört, sondern man kann auch kaum noch auf das ursprüngliche Problem zurückschließen.

Der übliche Weg zu solchem „instrumented code“ ist das Makro `assert` aus `<assert.h>`. Sein Name kommt von to assert = beteuern/versichern. Es stellt sicher, dass die als Argument übergebene Bedingung wahr ist. Wenn nicht, wird der Programmablauf unterbrochen und man kann wählen, ob man abbricht (abort), in den Einzelschritt-Debugger geht (debug) oder das Programm einfach weiterlaufen lässt (ignore).

Mit `assert` schreibt man insbesondere Vor- und Nachbedingungen [pre- and postconditions] für Funktionen:

9



Solche Assertions helfen, Fehler schneller zu finden. Sie machen obendrein aber auch klarer, was der Programmcode eigentlich tun soll: „design by contract“.

Assertions sind zum Finden echter Programmierfehler gedacht. Wenn dagegen der Benutzer eine Falscheingabe macht oder eine Datei nicht geöffnet werden kann, sollte das Programm darauf freundlich reagieren. In C ist eine übliche Lösung für solche Fälle, dass Funktionen, die fehlschlagen können, eine Fehlernummer zurückgeben – oder den Wert 0, wenn alles geklappt hat. In fortgeschritteneren Sprachen verwendet man dafür Exceptions.

Im ausgelieferten Programm kann man `assert` abschalten, um den Code schlanker und schneller zu machen. Dieses Makro wird *nicht* einkompiliert, wenn das Symbol `NDEBUG` definiert ist (Options > C/C++ compiler > Preprocessor > Defined symbols). Die meisten Entwicklungsumgebungen erlauben mehrere Sätze an Einstellungen zum Compilieren und Linken [build configurations]. Die Einstellungen lassen sich für jede Build Configuration getrennt wählen – der Typ des Chips, die Wahl zwischen Simulator und echter Hardware ebenso wie Optimierung und vordefinierte Symbole. Ab Werk gibt es bereits die Build Configurations „Debug“ und „Release“. Für die Release Build Configuration stellt man üblicherweise die Optimierung auf Maximum und definiert das Symbol `NDEBUG`. (Anders, als die Anleitung behauptet, macht die IAR Embedded Toolbench Letzteres nicht automatisch.) Achtung: Das Starten des Projekts klappt nur, wenn Options > Linker > Output > Format auf „Debug information“ steht.

Alle diese Maßnahmen helfen nichts ohne systematisches Testen. Jeder Pfad des Codes muss mindestens einmal durchlaufen worden sein, am besten zigfach. Nach jeder Änderung am Programm – zum Beispiel, wenn man einen Fehler beseitigt hat – muss man wieder alles testen. Das ist zu Fuß kaum zu machen. Deshalb testet man möglichst automatisch. Am besten *vor* dem Schreiben der ersten Programmzeile legt man Testfälle und die jeweils gewünschten Ergebnisse fest („test-driven development“). Diese lässt man für jedes Modul der Software (daher „unit testing“) automatisch durchprüfen. In Kern schreibt man dabei ein Hauptprogramm, das die Funktionen des Moduls aufruft und mit `assert` prüft, ob sie das jeweils gewünschte Ergebnis haben. Mit Hilfe von Bibliotheken wie CUnit (Link) geht das etwas komfortabler.