Programmierkurs
Vorlesung 7
Debugging

Andreas Naumann

Institut für Wissenschaftliches Rechnen
Universität Heidelberg

04. April 2023

Fehlersuche

- Compilerwarnungen
- Debugger
- Sanitizers

Programme haben Bugs

Jeder von Ihnen hat schon einmal einen Fehler in einem eigenen Programm gesucht.

Wie sind Sie das angegangen?

Methoden zur Fehlersuche

Fehlersuche zur Laufzeit

- Einfügen zusätzlicher Ausgaben (`printf()`-Debugging)

Methoden zur Fehlersuche

Fehlersuche zur Laufzeit

- Einfügen zusätzlicher Ausgaben (`printf()`-Debugging)
- Debugger

Methoden zur Fehlersuche

Fehlersuche zur Laufzeit

- Einfügen zusätzlicher Ausgaben (`printf()`-Debugging)
- Debugger
- überwachte Programmausführung (Sanitizers, Valgrind)

Methoden zur Fehlersuche

Fehlersuche zur Laufzeit

- Einfügen zusätzlicher Ausgaben (`printf()`-Debugging)
- Debugger
- überwachte Programmausführung (Sanitizers, Valgrind)

Fehlersuche zur Compilezeit

- Diagnose durch den Compiler (Warnungen)
- auch als statische Programm-Analyse bezeichnet
- In manchen Sprachen sehr weitgehend bis zu mathematischen Korrektheitsbeweisen

Der Compiler will helfen

- Moderne Compiler haben eine Vielzahl an Warnungen
- Standardsatz mit `-Wall`
- Bei Fehlern eventuell `-Wextra`
- Manpage zum Finden einzelner Warnungen
- Warnungen lesen, verstehen und beheben!

Compilerwarnungen: Beispiel

```
#include <iostream>

int fibonacci(int i) {
  switch(i) {
    case 0: return 0;
    case 1: return 1;
    default: fibonacci(i-1) + fibonacci(i-2);
  }
}

int main() {
  std::cout << fibonacci(1) << std::endl;
  std::cout << fibonacci(2) << std::endl;
  std::cout << fibonacci(3) << std::endl;
}
```

Was macht dieses Programm?

Compilerwarnungen: Beispiel

Der Compiler kann hier helfen:

```
g++ -Wall fibonacci.cc
```

Ausgabe:

```
fibonacci.cc:5:33: warning: expression result unused [-Wunused-value]
        default: fibonacci(i-1) + fibonacci(i-2);
                 ~~~~~~~~~~~~~~ ^ ~~~~~~~~~~~~~~
fibonacci.cc:7:5: warning: control may reach end of non-void function [-Wreturn-type]
    }
    ^
2 warnings generated.
```

Annahmen - Assertions

Oft ist es hilfreich implizierte Annahmen zu überprüfen:

```
1  #include <cassert>
2
3  double wurzel(double x) {
4          assert(x > 0 && "x muss größer 0 sein!")
5
6          // ...
7  }
```

- assert(...) Sorgt dafür, dass das Programm abstürzt wenn die Bedingung in den Klammern nicht erfüllt ist.
- Ver-Undung mit Fehlernachricht sorgt für bessere Lesbarkeit.
- Assertions sind nur in Debug-Builds aktiv.

Debugger

Debugger sind mächtige Werkzeuge zur Laufzeit-Untersuchung

- Breakpoints können das Programm unterbrechen
  - beim Erreichen bestimmter Codezeilen
  - beim Aufrufen bestimmter Funktionen
  - optional mit Bedingungen (z.B. `x > 0`)
  - beim Zugriff auf Variablen (Watchpoints)
  - beim Werfen oder Fangen von Exceptions
- Im angehaltenen Programm können Informationen ausgegeben und verändert werden
  - Werte von Variablen
  - Die aktuelle Funktionshierarchie (Backtrace)
  - Arbeitsspeicher-Inhalte
  - Werte von CPU-Registern

Wichtig

Debugger benötigen zusätzliche Informationen über das Programm (Option `-g3`, CMake: Debug-Build)

Standard-Debugger

GDB ist der Standard-Debugger unter Linux

- unterstützt viele Programmiersprachen
- weitreichende Dokumentation im Internet
- viele IDEs können mit GDB zusammenarbeiten
- erlaubt Rückwärts-Debugging
- erfordert spezielle Konfiguration unter macOS

LLDB ist der Standard-Debugger unter macOS

- Teil des LLVM-Projekts, das auch clang entwickelt
- schlecht dokumentiert
- Kommandozeilen-Interface oft sehr umständlich
- versteht Quellcode sehr gut durch Interaktion mit clang

Typische Benutzung eines Debuggers

- Programm stürzt ab
  $\rightarrow$ Ausführen mit Debugger, untersuchen nach Fehlerquelle
- Programm tut nicht was es soll
  $\rightarrow$ Setzen von breakpoints an strategischen Stellen im Code
- In beiden Fällen nähert man sich idealerweise Schrittweise der Fehlerquelle.

GDB: Tutorial

Vorbereitung (für alle Debugger)

Programm ohne Optimierung und mit Debug-Informationen kompilieren

- Kommandozeile: `-O0 -g3`
- CMake: Neues Build-Verzeichnis mit `-DCMAKE_BUILD_TYPE=Debug`

Speichern der eingegebenen Befehle zwischen Sitzungen aktivieren:

```
echo "set history save on" > ~/.gdbinit
```

Eine gute Übersicht und Vergleich der Befehler für GDB und LLDB gibt es hier:
https://lldb.llvm.org/use/map.html

GDB: Programm starten und Breakpoints

GDB für Programm `buggy` starten

```
gdb buggy
```

Breakpoint in Zeile 42 von `buggy.cc` setzen

```
break buggy.cc:42
```

Programm starten (oder neu starten, falls es läuft)

```
run [eventuelle Kommandozeilen-Argumente]
```

GDB: Programmsteuerung

Programm nach einem Breakpoint weiterlaufen lassen

```
continue
```

Nächste Zeile ausführen (nicht in Funktionen hineinspringen)

```
next
```

Nächste Zeile ausführen (in Funktionen hineinspringen)

```
step
```

Bis zum Ende der aktuellen Funktion weiterlaufen lassen

```
finish
```

GDB: Informationen ausgeben

Einen Ausdruck ausgeben (Variablen etc.)

```
print point.x
```

Alle lokalen Variablen anzeigen

```
info locals
```

Alle Funktionsargumente anzeigen

```
info args
```

Aktuellen Stacktrace anzeigen

```
backtrace # oder kurz bt
```

Der Stacktrace listet die ineinander geschachtelten Funktionsaufrufe auf, beginnend mit der aktuellen Funktion bis `main()`.

GDB: Fazit

Mächtiges Werkzeug, aber nicht leicht zu bedienen

GDB: Fazit

Mächtiges Werkzeug, aber nicht leicht zu bedienen

$\Rightarrow$ Einfacher mit Hilfe einer IDE

Beispiele:

- QT Creator
- DDD

Sanitizers

Sanitizer sind spezielle Compiler-Komponenten, die den generierten Code instrumentieren, so dass bestimmte Fehler zur Laufzeit erkannt werden:

- Address Sanitizer (`-fsanitize=address`): Erkennt ungültige Speicherzugriffe
- Undefined Behavior Sanitizer (`-fsanitize=undefined`): Erkennt Programmcode, der undefiniertes Verhalten auslöst. Nicht sehr zuverlässig.
- Thread Sanitizer (`-fsanitize=thread`): Erkennt Probleme bei der Programmierung mit mehreren Threads. Hier können mehrere CPU-Kerne gleichzeitig auf eine Variable zugreifen, was sogenannte data races erzeugt.
- In eine ähnliche Kategorie fällt das externe Tool valgrind, mit dem das Programm auf simulierter Hardware ausgeführt wird, die viele Fehler erkennt, aber durch die Simulation sehr langsam ist.

Warum Sanitizers?

- Oft kann ein Programm nach einem falschen Speicherzugriff etc. noch lange weiterlaufen.
- Viel später greift ein anderer Programmteil auf den überschriebenen Speicherbereich zu, und das Programm crasht.
- Ein Debugger kann nur sagen, wann dieser Crash passiert.
- Ein Sanitizer erkennt oft den wirklichen Grund für einen Bug.
- Manche Klassen von Fehlern treten in Debug-Builds nicht auf und können mit Debuggern nicht untersucht werden.
- Sanitizers können auch mit Optimierung verwendet werden, man muss nur die Debuginformation anschalten (`-g3`).

Weitere Informationen

Das Thema Sanitizer und Code Instrumentierung ist sehr umfangreich. Diese Vorlesung kann Ihnen daher nur die Existenz solcher Tools vermitteln.

- Weitere Informationen zu den GCC instrumentierungen finden Sie hier:
  https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html
- Weitere Informationen zu Valgrind finden Sie hier:
  https://valgrind.org/docs/manual/manual-core.html

Zusammenfassung

Beim Programmieren wird es früher oder später zu Fehlern kommen. Um diese leichter zu finden, können folgende Tools verwendet werden:

- Compiler Warnings: `-Wall -Wextra`
- Assertions: `assert(<Condition> && "Message")`
- Debugger: gdb oder lldb
  - `gdb --args <binary> arg1 arg2 arg3`
  - `break <sourcefile>:<line>`
  - `info locals`
  - `print point.x`
  - `backtrace`
- Sanitizers:
  - `g++ -fsanitize=...`
  - `valgrind --tool=memcheck <binary>`
- Profiler
  - valgrind, callgrind & kcachegrind
  - `g++ -fprofile` und gprof
  - valgrind, massif & `ms_print`