Programmierwerkzeuge

Projektmanagement im Softwarebereich – SeqAn

David Weese
April 2009

Inhalt

- Build Sytem | **make**
- Debugger | **gdb, ddd**
- Entwicklungsumgebung | **Visual Studio**
- Profiler | **gprof**
- Memory Debugger | **valgrind**
- Weitere Tools | **doxygen, svn**
- Bug Tracker | **trac**

Einige Folien entstammen dem Kurs von Matthias Neubauer:
http://sommercampus2004.informatik.uni-freiburg.de/ProgrammierwerkzeugeKurs

BUILD SYSTEM

Allgemeines

Programmbau mit Make:

- Änderungen implizieren oft mehrere Zwischenschritte, um das Programm/Produkt zu bauen:
  - Übersetzen von Quelldateien in Objektdateien
  - Binden von Objektdateien zu ausführbaren Programmen
  - Erzeugen der Dokumentation aus den Quelldateien
- Zwischenschritte können von anderen abhängen (Abhängigkeitsgraph)

Makefiles definieren:

- welche Komponenten es gibt
- wovon sie abhängen
- Schritte zur Konstruktion der Komponenten

Makefile: Regeln

- für eine oder mehrere Komponenten
- Abhängigkeiten
- Befehle

```
ziel1 ziel2 … zieln: quelle1 quelle2 … quellem
    kommando1
    kommando2
    kommando3
    …
```

* Kommandos müssen mit **TABS** eingerückt sein!

Beispiel: duden

- Programm duden besteht aus zwei Komponenten: grammatik.c und woerterbuch.c
- Für beide Komponenten: C-Quelldatei wird mit Hilfe von cc in Objektdatei übersetzt
- Binden der Objektdateien zum ausführbaren Programm

Zu tun wäre:

```
cc grammatik.c -c -o grammatik.o
cc woerterbuch.c -c -o woerterbuch.o
cc grammatik.o woerterbuch.o -o duden
```

Makefile für duden

```
duden: grammatik.o woerterbuch.o
	cc grammatik.o woerterbuch.o -o duden

grammatik.o: grammatik.c
	cc grammatik.c -c -o grammatik.o

woerterbuch.o: woerterbuch.c
	cc woerterbuch.c -c -o woerterbuch.o

clean:
	rm grammatik.o woerterbuch.o duden
```

oder kürzer (implizite Regeln)

```
duden: grammatik.o woerterbuch.o
        cc grammatik.o woerterbuch.o -o duden

clean:
        rm grammatik.o woerterbuch.o duden
```

Funktionsweise von **make**

- Berechne Abhängigkeitsgraph
- Für Zielkomponente A
  - bestimme Komponenten A_1, …, A_n von denen A abhängt
  - rufe Algorithmus für alle A_i rekursiv auf
  - falls A nicht existiert, oder ein A_i neu gebaut/verändert wurde: erzeuge A mit Kommandos
- Erkennen von Änderungen einer Datei
  - Datum der letzten Änderung wird verwaltet
  - Datei geändert, falls jünger als von ihr abhängige Komponente

Features von **make**

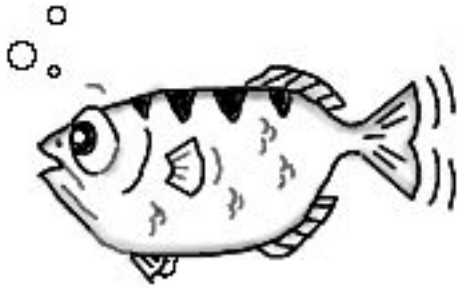
- Variablen `var=wert`
- Referenzierung durch `$(var)`
- Implizite Regeln
  - Definition von Standardregeln für Komponenten mit best. Namensmuster
  - Vordefiniert ist bspw:

```
.c.o:
    $(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
```

- Implizite Variablen für erstes Ziel `$@` oder Quelle `$<`
- If-Anweisungen und Makros

GNU Make Manual - http://www.gnu.org/software/make/manual/

DEBUGGER



Fehlersuche mit **gdb** und **ddd**

- Ein Debugger führt ein Programm kontrolliert aus
  - Programm in definierter Umgebung ausführen
  - Programm unter bestimmten Bedingungen anhalten lassen
  - Zustand eines angehaltenen Programms untersuchen
  - Zustand eines angehaltenen Programms verändern
- GNU Debugger **gdb**
  - interaktives Programm mit Kommandozeilensteuerung
- Data Display Debugger **ddd**
  - graphische Benutzeroberfläche zu gdb

Programmbeispiel:

- Es sollen Zahlen von der Kommandozeile eingelesen, sortiert und wieder ausgegeben werden

```c
#include <stdio.h>
#include <stdlib.h>

void shell_sort(int a[], int size) {
   int i, j;
   int h = 1;

   do {
      h = h * 3 + 1;
   } while (h <= size);

   do {
      h /= 3;
      for (i = h; i < size; i++) {
         int v = a[i];
         for (j = i; j >= h && a[j-h] > v; j -= h)
            a[j] = a[j - h];
         if (i != j)
            a[j] = v;
      }
   } while (h != 1);
}
```

```c
int main (int argc, char *argv[]) {
   int *a;
   int i;

   a = (int *) malloc((argc - 1) * sizeof(int));
   for (i = 0; i < argc - 1; i++)
      a[i] = atoi(argv[i+1]);

   shell_sort(a, argc);

   for (i=0; i < argc - 1; i++)
      printf ("%d ", a[i]);
   printf ("\n");

   free (a);
   return 0;
}
```

Beispielprogramm sample.c

Ausführung von sample

- Manchmal geht's:

```
$ ./sample 1 8 5 3 4 7
1 3 4 5 7 8
$
```

- und manchmal nicht:

```
$ ./sample 1 8 5 3 4
0 1 3 4 5
$
```

Programme debuggen mit **ddd**

- Übersetzen mit *Debugging-Informationen*:

```
$ cc –g sample.c –o sample
$
```

- Starten der Debugging-Sitzung:

```
$ ddd ./sample&
$
```

- Breakpoint in Zeile 31 und **run** mit **1 8 5 3 4** als Kommandozeile
- *View->Data Window* anzeigen
- Im *Data Window* Rechtsklick *New Display* und ***a @ 6** hinzufügen
- Schrittweise debuggen

Debugger steuern

Ausführung steuern:

- **Run** startet Debugger
- **Step** führt einzelne Zeile aus und springt in Subroutinen
- **Next** führt einzelne Zeile aus und überspringt Subroutinen
- **Until** springt aus Schleifen raus
- **Finish** springt aus Subroutinen zurück zum Aufrufer

Variablen/Ausdrücke anzeigen:

- Variable anzeigen `print i`
- Arrayelement anzeigen `print a[3]`
- Die ersten 6 Elemente eines Arrays anzeigen `print a[0]@6`

Ausdrücke können als *New Display* im *Data Window* hinzugefügt werden

Siehe da!

shell_sort(a, argc) muss zu

shell_sort(a, argc – 1) zu geändert werden.

Funktionsweise des **gdb**

- Betriebssystem erlaubt Kontrolle der Programmausführung
- Verbindung zwischen Programmspeicher und Originalquelltext
  - Debugging-Informationen enthalten Symbolnamen, Typinfos und Zeilennummern

```
$ gdb –S –g sample.c
$ less sample.s
…
.globl main
        .type   main, @function
main:
.LFB6:
        .loc 1 26 0
```

← **main** beginnt in Zeile 26

Features von Debuggern

- Ändern der Programmausführung
  - Die Kommandos return und jump
- Ändern des Programmcodes
- Post-Mortem-Debugging
  - Untersuchen des letzten Zustands vor Programmabsturz

```
$ gdb ./sample core
```

gdb Documentation - http://sourceware.org/gdb/documentation/

ddd Documentation - http://www.gnu.org/manual/ddd/

ENTWICKLUNGSUMGEBUNG

Allgemeines

Frei verfügbare C++ IDEs:

- Microsoft Visual Studio Express (www.microsoft.com)
- Eclipse (www.eclipse.org)
- Kdevelop (www.kdevelop.org)
- Xcode (developer.apple.com/tools/xcode/)
- Emacs, Anjuta, ...

IDE (Integrated Development Environment) besteht aus:

- Texteditor
- Compiler
- Linker
- Debugger

Visual Studio

- Express-Version für alle frei verfügbar
- VS 2003, VS 2005, VS 2008 frei für Studenten über FU-MSDNAA

Wie kann man mit VS:

- ein neues Projekt anlegen
- Dateien hinzufügen
- das Projekt kompilieren
- den Debugger benutzen

FU-MSDNAA - https://msdnaa.mi.fu-berlin.de/

MSDN Visual C++ Reference - Übersicht, Language Reference

PROFILER

Laufzeitanalyse mit **gprof**

- Profiler - Programmierwerkzeuge, die das Laufzeitverhalten von Software analysieren
- Tuning: systematische Steigerung der Leistung eines Programms
- Laufzeitanalyse
  - an welchen Stellen sind Leistungssteigerungen möglich
  - was sind die Effekte einer Optimierung

GNU Profiler **gprof**

- wertet Programmprofile aus
- Programm erstellt Programmprofil während des Programmablaufs
- Programmprofil gibt für jede Funktionen an
  - wie oft ausgeführt
  - wie lange ausgeführt

So geht‘s

- Übersetzen mit *eingerichteter Profilierung*:

```
$ g++ -pg sample.cpp -o sample
$
```

- Starten des Programms, Profil wird in Datei **gmon.out** geschrieben:

```
$ ./sample
$ 100000 ints read
$
```

- Analysieren des Profils mit **gprof**:

```
$ gprof sample
$
```

Das flache Profil

- gibt an, wie sich die Laufzeit auf die einzelnen Funktionen verteilt:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 69.07      0.11     0.11        1   110.51   110.51  shell_sort(int*, int)
 31.40      0.16     0.05        1    50.23    50.23  quick_sort(int*, int,
  0.00      0.15     0.00        1     0.00     0.00  global constructors
  0.00      0.15     0.00        1     0.00     0.00  __static_initializati
```

Das strukturierte Profil

- Gibt für jede Funktion *f* an
  - Anteile von *f* und der von f aufgerufenen Funktionen an der Gesamtlaufzeit (`%time`)
  - von welcher Funktion *f* aufgerufen wurde
  - welche Funktionen *f* aufgerufen hat

C/C++-Programme optimieren mit dem Profiler gprof (linuxfocus.org)

gprof Documentation - http://sourceware.org/binutils/docs/

```
granularity: each sample hit covers 2 byte(s) for 6.22% of 0.16 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    0.16                 main [1]
                0.11    0.00       1/1           shell_sort(int*, int) [2]
                0.05    0.00       1/1           quick_sort(int*, int, int) [3]
-----------------------------------------------
                0.11    0.00       1/1           main [1]
[2]     68.7    0.11    0.00       1         shell_sort(int*, int) [2]
-----------------------------------------------
                                154478           quick_sort(int*, int, int) [3]
                0.05    0.00       1/1           main [1]
[3]     31.2    0.05    0.00       1+154478  quick_sort(int*, int, int) [3]
                                154478           quick_sort(int*, int, int) [3]
-----------------------------------------------
                0.00    0.00       1/1           __do_global_ctors_aux [12]
[10]     0.0    0.00    0.00       1         global constructors keyed to main [1
                0.00    0.00       1/1           __static_initialization_and_dest
-----------------------------------------------
                0.00    0.00       1/1           global constructors keyed to mai
[11]     0.0    0.00    0.00       1         __static_initialization_and_destruct
-----------------------------------------------
```

MEMORY DEBUGGER

Speicheranalyse mit **valgrind**

- Valgrind ist ein leistungsfähiges Toolset für
  - Profiling
  - Memory Debugging
  - Memory/Cache Profiling
  - Thread Debugging
- Memory Debugger – sucht Fehler im Speicher-Management von Programmen
- Zu testendes Programm mit Debugging-Informationen übersetzen. (Parameter "-g" beim gcc)

So geht’s

- Übersetzen mit *Debugging-Informationen*:

```
$ g++ -g example.cpp -o example
$
```

- Starten der Debugging-Sitzung:

```
$ valgrind --leak-check=yes ./example 3
$
```

Grenzüberschreitung ...

- Fehlerhafter Zugriff jenseits der Array-Grenzen

  Auszug aus example.cpp:

```
45:  int *i = new int[10];
46:  i[10] = 13;
47:  cout << i[-1] << endl;
48:  delete[] i;
```

... ergibt

```
==17056== Invalid write of size 4
==17056==    at 0x401026: test_3() (example.cpp:46)
==17056==    by 0x401159: main (example.cpp:135)
==17056==  Address 0x51E9058 is 0 bytes after a block of size 40 alloc'd
==17056==    at 0x4A1B858: malloc (vg_replace_malloc.c:149)
==17056==    by 0x401019: test_3() (example.cpp:45)
==17056==    by 0x401159: main (example.cpp:135)
==17056==
==17056== Invalid read of size 4
==17056==    at 0x401034: test_3() (example.cpp:47)
==17056==    by 0x401159: main (example.cpp:135)
==17056==  Address 0x51E902C is 4 bytes before a block of size 40 alloc'd
==17056==    at 0x4A1B858: malloc (vg_replace_malloc.c:149)
==17056==    by 0x401019: test_3() (example.cpp:45)
==17056==    by 0x401159: main (example.cpp:135)
0
==17056==
==17056== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 8 from 1)
==17056== malloc/free: in use at exit: 0 bytes in 0 blocks.
==17056== malloc/free: 1 allocs, 1 frees, 40 bytes allocated.
==17056== For counts of detected errors, rerun with: -v
==17056== All heap blocks were freed -- no leaks are possible.
```

Fehlerarten

Valgrind erkennt

- Illegale Zugriffe:
  - Speicher wurde nicht initialisiert
  - Zugriff außerhalb reservierten Speichers
- Allocation/Deallocation Mismatches:
  - mit **new** alloziert und mit **free (**anstatt **delete**) freigegeben
  - Feld mit **delete (**anstatt **delete[]**) freigegeben
- Memory Leaks – nicht freigegebener Speicher
- Double Frees – doppelt freigegebener Speicher

Funktionsweise von **valgrind**

- Ist virtuelle Machine mit JIT
- Übersetzt Binärcode des Programms in plattform-unabhängigen Byte-Code
- Dieser sog. Ucode wird von Valgrind-Tools modifiziert
- Danach rückübersetzt und ausgeführt

- Dadurch lassen sich beliebige Programme analysieren
- Laufzeit ist aber um ein Vielfaches größer

Valgrind Documentation - http://valgrind.org/docs/

WEITERE TOOLS

Dokumentieren ...

- Das Verwalten und Modifizieren von großen C++ Paketen kann sehr komplex sein
- Gute Dokumentation wird dann sehr wichtig
- Dokumentiert werden müssen
  - Funktionen, Funktionsparameter und Return-Werte
  - Klassen und Member-Variablen

Doxygen

- Ist ein Werkzeug zur Dokumentation von C++ Quelltexten ähnlich javadoc
- Analysiert Quelltextkommentare im Doxygenformat
- Erzeugt html-, pdf-, latex-, … Dateien mit der entsprechenden Dokumentation

... mit **Doxygen**

```
* Beispiel einer Dokumentation einer Funktion
/** Dokumentation of a function
*  @paramlower lower bound of the range
*  @paramupper upperbound of the range
*  @return A vector with the same size as this
*            vector and binary elements
*  @warning some detail ..
*  @todo ..
*  @bug                …
**/
FVector findInRange(float lower, float upper) const;
```

Doxygen - http://www.stack.nl/~dimitri/doxygen/

Doxygen-Beispiel - http://api.kde.org/4.x-api/kdevplatform-apidocs/

Zusammenarbeiten ...

Bei großen Projekten treten folgende Probleme auf:

- Wie koordiniert und synchronisiert man Code zwischen verschiedenen Entwicklern?
- Wie sichert man Code gegen versehentliches Löschen oder archiviert alte Stände?

Per Mail? Dateien online ablegen? Auf Netzlaufwerken arbeiten?

Nein!

Lösung: **Source Code Management Tool** (SCM) bspw.:

- Subversion
- CVS
- Trac
- Git

... mit einem SCM

SCM besteht aus:

- **Repository** – Code gespeichert auf zentralem Server
- **Working copy** – Entwickler/-in checkt eine Kopie des Repositories auf seinen/ihren Computer aus
- **Revision history** – Jede Änderung einer Datei wird auf dem Server gespeichert, kann auch wieder rückgängig gemacht werden
- **Conflict Handling** – Was passiert wenn zwei Entwickler dieselbe Datei verändern? In der selben Zeile?

Abläufe

1. Entwickler checkt **Repository** initial aus
2. Ändert seine lokale **Working copy**
3. **Commited** seine Änderungen in das **Repository**
4. **Updated** seine **Working copy** mit den Änderungen der anderern Entwickler
5. Datei-Differenzen (**diffs**) werden zwischen **Repository** und **Working copies** hin- und hergeschickt

Differenzen (Patches) zweier Dateien liefert das Tool **diff**

```
diff –u Datei1 Datei2 > MeinBugFix
```

Anwenden kann man diese mit **patch**

```
patch –p0 < MeinBugFix
```

Subversion - SVN

- Kommandos für Kommunikation mit dem Server:
  - `svn checkout <URL>`
  - `svn update`
  - `svn commit`
- Offline Kommandos
  - `svn add <Datei>`
  - `svn delete <Datei>`
  - `svn diff`
  - `svn rename`
  - `svn move`
  - …
  - `svn help <Kommando>`

TortioseSVN (Windows Client) - http://tortoisesvn.net/ und Tutorial

SVN Homepage - http://subversion.tigris.org/ und Tutorial

ENDE