Programmieren mit den UNIX-Rechnern

M. Jäger – FB MNI

11. Oktober 2004

Dieser Artikel ist ein kurze Einführung für diejenigen, die erstmals mit den UNIX-Systemen des FB MNI Programme entwickeln.

Zum Entwicklungssystem gehören unter anderem

- C-Präprozessor
- C-Compiler
- C++-Compiler
- Linker
- Make
- Debugger
- Versionsverwaltung
- Laufzeitmessung

1 Präprozessor, Compiler und Linker

Der Programmerzeugungsprozess besteht aus mehreren Schritten:

- Der Präprozessor bearbeitet die Präprozessor-Direktiven im Quelltext, z.B. `#include`- oder `#define`-Anweisungen
- Der Compiler erzeugt ein Objektmodul
- Der Linker erzeugt aus einem oder mehreren Objektmodulen ein lauffähiges Programm. Die Objektmodule liegen entweder als einzelne Dateien (Suffix: *.o*) oder in Form von Objektmodul-Bibliotheken (*lib*...) vor.

Das bei uns installierte Entwicklungssystem basiert auf dem GNU-Compiler. Die beteiligten Programme (Präprozessor, Compiler und Linker) werden über ein gemeinsames Kontrollprogramm bedient:

- `gcc` für C-Programme
- `g++` für C++-Programme

Verwendungsbeispiele:

1. Ein nicht modularisiertes C++-Programm `a.out` aus dem Quelltext `uebung-1.cc` erzeugen:

   ```
   g++ uebung-1.cc
   ```

2. Wie oben, aber das lauffähige Programm soll `uebung-1` statt `a.out` heißen:

   ```
   g++ -o uebung-1 uebung-1.cc
   ```

3. Wie oben, aber das lauffähige Programm soll die Verwendung symbolischer Namen beim Debuggen erlauben:

   ```
   g++ -g -o uebung-1 uebung-1.cc
   ```

4. Ein modularisiertes C++-Programm `uebung-2` aus den Quellen `modul-1.cc` und `modul-2.cc` erzeugen :

   ```
   g++ -c modul-1.cc
   g++ -c modul-2.cc
   g++ -o uebung-2 modul-1.o modul-2.o
   ```

   `g++ -c xyz.cc` erzeugt aus dem Quelltext ein Objektmodul `xyz.o`. Das 3. Kommando aktiviert den Linker.

5. Wie oben, aber das lauffähige Programm soll die Verwendung symbolischer Namen beim Debuggen erlauben:

   ```
   g++ -g -c modul-1.cc
   g++ -g -c modul-2.cc
   g++ -g -o uebung-2 modul-1.o modul-2.o
   ```

6. Der Linker sucht die Standardfunktionen ohne weiteres Zutun in den Standardbibliotheken. Falls besondere Bibliotheken benötigt werden (etwa die POSIX-Thread-Bibliothek), müssen diese explizit angegeben werden.

   Beispiel: Dem Linker mitteilen, dass er die Nonstandard-Bibliothek *libpthread* durchsuchen soll.

   ```
   g++ -o uebung-2 modul-1.o modul-2.o -lpthread
   ```

2 Make

Make ist ein sprachunabhängiges Dienstprogramm, dass die Erzeugung von Programmen (oder Dokumenten) aus verschiedenen Quellmodulen unterstützt. Es gibt mehrere Varianten davon, wir verwenden das Programm *gmake* (GNU-Make).

Grundprinzip: In einer Datei namens *Makefile* definiert man

- Zieldateien (Dateien, die man gerne generieren möchte)
- dazu direkt benötigte Quelldateien
- Aktionen, mit denen die Zieldateien aus den Quelldateien erzeugt werden

Beim Aufruf gibt man den Zieldateinamen an: `make zieldatei`. Make erzeugt das Ziel.

Beispiel-Makefile:

```
uebung-2: modul-1.o modul-2.o
     g++ -o uebung-2 modul-1.o modul-2.o

modul-1.o: modul-1.cc modul-1.h uebung-2.h
     g++ -c modul-1.cc

modul-2.o: modul-2.cc modul-2.h modul-1.h
     g++ -c modul-1.cc
```

Achtung: Aktionszeilen, wie die `g++`-Aufrufe beginnen mit einem `Tabulator`-Zeichen!

Bei der Zieldatei-Erzeugung berechnet *make* aus den direkten Abhängigkeiten die indirekten Abhängigkeiten und berücksichtigt Modifikations-Zeitstempel der Dateien, um überflüssige Generierungsschritte zu vermeiden.

Beispiel-Szenario:

- Der Benutzer erzeugt mit `make uebung-2` ein Programm, testet es findet einen Fehler und korrigiert daraufhin eine Funktionsimplementierung in `modul-1.cc`.
- Der Benutzer ruft danach erneut *make* auf: `make uebung-2`.
- Make prüft, ob die für das Ziel benötigten Quellen vorhanden und auf einem aktuellen Stand sind. Dies beinhaltet die direkt benötigten Quellen (`modul-1.o`, `modul-2.o`) und auch die indirekt benötigten (`.cc`- und `.h`-Dateien).
- Make stellt fest, dass `modul-1.cc` einen neueren Modifikationszeitpunkt hat als `modul-1.o`. `modul-1.o` muss also aktualisiert werden. Damit ist auch `uebung-2` nicht mehr aktuell!
- Make veranlasst die erforderlichen Aktionen:

```
g++ -c modul-1.cc
g++ -o uebung-2 modul-1.o modul-2.o
```

- `modul-2.cc` muss dagegen nicht neu übersetzt werden.

2.1 Default-Datenbasis

Immer wieder benötigte Erzeugungsregeln kann man auch in Form von Erzeugungsschablonen beschreiben, etwa den Erzeugungsprozess eines Objektmoduls aus einem C++-Quelltextmodul.

Make benutzt eine Reihe von Standardschablonen. Dadurch kann man im Makefile auf die Beschreibung von Standardabhängigkeiten und Standard-Aktionen verzichten. (Im einfachsten Fall kann man komplett auf ein *Makefile* verzichten).

Beispiele:

1. Ein nichtmodularisiertes Programm kann statt eines direkten Compileraufrufs

   ```
   g++ -o uebung-1 uebung-1.cc
   ```

   auch einfach mit

   ```
   make uebung-1
   ```

   erzeugt werden. Ein Makefile ist dazu nicht nötig. Make sucht nach möglichen Quellen (C, C++, Fortran, Assembler, Pascal usw.) und findet dabei die `C++`-Quelltextdatei `uebung-1.cc`. Für die Erzeugung existiert eine Erzeugungsschablone.

2. Für unser 2-Modul-Beispiel `uebung-2` ist ein Makefile erforderlich, denn woher sollte *make* sonst wissen, welche Module zu dem Programm gehören. Es reicht aber, die Abhängigkeiten darin anzugeben:

   ```
   uebung-2: modul-1.o modul-2.o
   modul-1.o: modul-1.cc modul-1.h uebung-2.h
   modul-2.o: modul-2.cc modul-2.h modul-1.h
   ```

   Die Aktionen stehen nicht im Makefile, also verwendet *make* die Standardaktionen.

2.2 Parameter in Standardschablonen

In den Make-Standardschablonen werden redefinierbare Parameter zur Spezifikation der Aktionen benutzt. Die wichtigsten:

CC	Name des C-Compilers
CXX	Name des C++-Compilers
CFLAGS	C-Compiler-Optionen
CXXFLAGS	C++-Compiler-Optionen
LDFLAGS	Linker-Optionen
CPPFLAGS	Präprozessor-Optionen

Durch Redefinition können die Standardaktionen modifiziert werden.

Beispiele:

- Sie wollen den Debugger benutzen, und benötigen dazu die Compileroption `-g`. Wenn sie die *Make*-Standardregel benutzen wollen, müssen Sie `CXXFLAGS` redefinieren, entweder im *Makefile* oder in der Kommandozeile des *make*-Aufrufs.

  ```
  make uebung-1 CXXFLAGS=-g
  ```

- Sie haben im Compilerbaupraktikum ihren eigenen C++-Compiler programmiert und wollen diesen - mit einigen speziellen Aufrufoptionen - benutzen. *Makefile*:

  ```
  CXX=mein-eigener-c++-compiler
  CXXFLAGS=-schnell -fehlerfrei -cool

  uebung-2: modul-1.o modul-2.o
  modul-1.o: modul-1.cc modul-1.h uebung-2.h
  modul-2.o: modul-2.cc modul-2.h modul-1.h
  ```

3 Debugger und Debugger-Frontend

Wir benutzen den GNU-Debugger *gdb*, einen mächtigen Debugger, der neben „normalen“ Debugging-Möglichkeiten auch weitergehende Möglichkeiten bietet (z.B. Programm auf einem entfernten Rechner kontrollieren, Kontrolle über einen schon gestarten Prozess übernehmen).

GDB ist über Kommandozeilen bedienbar, die Kommandobeschreibung ist Online verfügbar (`help`-Kommando).

Zur bequemeren Bedienung mit der Maus gibt es graphische Benutzerschnittstellen, die als separate „Frontend“-Programme realisiert sind. Die Frontend-Programme reagieren auf Menü- und Toolbar-Selektionen und übermitteln an GDB entsprechende Kommandozeilen.

Als Debugger-Frontend wird `ddd` empfohlen. Die Bedienung sollte selbsterklärend sein.

Eine kleine Auswahl von `gdb`-Kommandos:

help	Auskunft
show args	Programmargumente anzeigen
run	Ausführung des Programms starten
break	Unterbrechungsstelle (*breakpoint*) definieren
watch	Wertänderung (Variable oder Ausdruck) überwachen
backtrace	aktive Funktionsaufruf zeigen
up, down	in der Funktionsaufruf-Hierarchie nach oben (unten) gehen
return	Rücksprung aus aktuellem Aufruf veranlassen
set variable	Variable modifizieren
print, print*	Werte anzeigen
continue	Ausführung fortsetzen
jump	Ausführung an einer bestimmten Stelle fortsetzen
until	Ausführung bis zu einer bestimmten Stelle fortsetzen
step	Einzelschritt(e) ausführen mit Verfolgung von Funktionsaufrufen
next	Einzelschritt(e) ausführen ohne Verfolgung von Funktionsaufrufen
finish	Ausführung bis zum Rücksprung aus dem aktuellen Funktionsaufruf
attach	Kontrolle über einen anderen Prozess übernehmen
set environment	Environment-Variable definieren
set args	Argumente definieren
thread	Thread wechseln
thread apply	Kommando auf Thread-Liste anwenden
apply all	Kommando auf alle Threads anwenden

4 Versionsverwaltung

Bei größeren Entwicklungsvorhaben ist die Verwaltung von Programmversionen unerlässlich. Hier soll nur eine kurze Anleitung zur *rcs*-Bedienung gegeben werden. RCS verwaltet verschiedene Versionen *einer Datei*. Aufbauend auf RCS kann das System CVS komplette Verzeichnisse mit allen zu einem Entwicklungsprojekt gehörenden Quelltexten verwalten.

RCS (Revision Control System) kann verschiedene Versionen einer Textdatei verwalten. Zu jeder Textdatei erstellt `rcs` eine Versionsdatei (RCS-Datei) mit folgendem Inhalt:

- die jeweils neueste Version der Textdatei
- die Differenzen zu den älteren Versionen
- Änderungskommentare des Entwicklers
- Versionsnummern

Die Versionsnummern sind zweistufig in *Revision* (größere Änderungen) und Version (kleinere Änderungen) aufgeteilt, z.B. 2.3 - Revision 2, Version 3.

- Mit dem „check in“-Kommando *ci* erzeugt man eine erste oder eine neue Version einer Datei
- Mit dem „check out“-Kommando *co* extrahiert man eine bestimmte Version aus der Versionsdatei:
  - zwecks Weiterentwicklung: `co -l`
    RCS sperrt die Version gegen anderweitige Modifikation.
  - für rein lesende Zugriffe: `co`
    z.B. zum Anschauen, Ausdrucken, Compilieren
- Mit dem `rcslog`-Kommando kann man Informationen über die Versionen anzeigen.

Beispiele:

`ci uebung-1.cc`	„einchecken“ – erste oder nächste Version erzeugen, Versionsdatei: `uebung-1.cc,v`
`co uebung-1.cc`	letzte Version zum Lesen extrahieren
`co -l uebung-1.cc`	letzte Version zum Weiterentwickeln extrahieren
`co -r2.3 uebung-1.cc`	Version 2.3 extrahieren
`rcs2log uebung-1.cc`	Versionsinfo anzeigen

5 Laufzeitprofile

Um Algorithmen effizient zu implementieren, benötigt man Messmechanismen, die beim Programmlauf für jeden Funktionaufruf Zeitmessungen durchführen und eine Funktionsaufruf-Statistik erstellen.

Die Erweiterung eines Programm mit Anweisungen, die solche Statistiken („Ausführungsprofile“, engl.: „execution profile“) erzeugen, übernimmt auf Wunsch der Compiler (Option: `-pg`).

Wenn man ein derart „instrumentiertes“ Programm anschließend ausführt, erzeugt es eine Datenbasis mit den Messwerten. Man verwendet ein passendes Reportprogramm, dass aus der Datenbasis dann Laufzeittabellen erstellt: `gprof`.

Beispiel:

```
$  make uebung1 CXXFLAGS=-pg    - Übersetzen mit Profil-Option -pg
$  uebung1                      - Ausführung erzeugt Profil: gmon.out
$  gprof uebung-1               - Auswertung und Ausgabe der Messergebnisse
```