ALBERT-LUDWIGS-UNIVERSITÄT FREIBURG
INSTITUT FÜR INFORMATIK
Lehrstuhl für Mustererkennung und Bildverarbeitung
Prof. Dr.-Ing. Hans Burkhardt

Georges-Köhler-Allee
Geb.052, Zi 01-029
D-79110 Freiburg
Tel. 0761 - 203 - 8260

Kurzdokumentationen der Tools zum Softwarepraktikum SS 2001

30. April 2001

Inhaltsverzeichnis

1 Einleitung

Wichtiges Lernziel des Softwarepraktikums ist Vertrautheit mit einigen Arten von Tools, die bei der Softwareentwicklung zum Einsatz kommen. In dieser Sammlung sollen einige Vertreter der wichtigsten dieser Werkzeuge kurz vorgestellt und die wesentlichen Anwendungsmöglichkeiten dargestellt werden. Referenzen auf ausführlichere Anleitungen sind am Schluß der jeweiligen Abschnitte angegeben.

2 g++ - GNU C++ Compiler

2.1 Kurzbeschreibung

Der `g++` wird benötigt, um aus den C++ Quelldateien ausführbare Programme zu erzeugen, oder um eigene Bibliotheken zu erstellen. Intern läuft die Compilierung eines Programms in vier Schritten ab, wovon man als Benutzer aber üblicherweise maximal zwei zu Gesicht bekommt. Diese Schritte sind:

1. **„preprocessing“** Im ersten Schritt werden die Präprozessor-Befehle[1] interpretiert. Außerdem entfernt der Präprozessor Kommentare aus dem Sourcecode. Das Ergebnis ist ein vorverarbeiteter C++-Source-Code.

2. **„compilation“** Die eigentliche Übersetzung des vorverarbeiteten C++-Source-Codes in Assembler. Das Ergebnis ein Assembler-Source-File, in dem die Assembler-Befehle im Klartext notiert sind.

3. **„assembly“** Übersetzung des Assembler-Source-Files in Maschinen-Code. Das Ergebnis ist ein Object-File, in dem sowohl der Maschinencode, als auch ein Symbol-Table, also eine Tabelle, die im wesentlichen die Namen der enthaltenen Funktionen und ihrer Startadressen im Maschinencode enthält. [2]

4. **„linking“** Zusammenhängen von mehreren Object-Files, Anhängen eines „Runtime“-Kopfes und der Standard-Bibliotheken, so daß ein ausführbares Programm entsteht.

[1] Die bekanntesten Präprozessor-Befehle sind `#include` (um andere Dateien, üblicherweise Header-Dateien, an dieser Stelle einzufügen), `#define` (zum Definieren von Makros), sowie `#if`, `#ifdef`, `#ifndef`, `#else` und `#endif` (für die bedingte Compilierung, z.B. Debugging-Code ein und auszuschalten, oder um den Source-Code auf mehreren Architekturen compilierbar zu machen.)

[2] Wichtig: Da bei C++ (im Gegensatz zu reinem C) mehrere Funktionen mit dem gleichen Namen, aber verschiedenen Parametern erlaubt sind, unterschieden sich diese Symbol-Namen, je nachdem, ob das Programm mit einem C oder C++ Compiler übersetzt wurde. Um eine mit einem reinen C-Compiler übersetzte Funktion (z.B. aus einer C-Bibliothek) von C++ aus aufzurufen, ist es daher nötig, den Funktionsprototyp innerhalb eines `extern "C" { ... }` Block zu definieren. Bei den Standard-Bibliotheken ist dieser Block immer schon im Header-File vorhanden.

2.2 Wichtigste Anwendungsvarianten

2.2.1 Programme, die nur die Standard-Bibliothek verwenden

Zum Übersetzen eines C++ Programms, daß nur aus einem Source-File besteht und das nur die Standard-Bibliotheken benutzt, reicht ein einfaches

```
g++ example.cc -o example
```

Dabei werden alle vier oben beschriebenen Schritte ausgeführt und es ensteht das ausführbare Programm „example“. Ist der Sourcecode über mehrere Files verteilt (was auch schon bei kleineren Projekten unbedingt anzuraten ist) kann man diese mit

```
g++ example.cc classes1.cc classes2.cc -o example
```

übersetzen. Das wird ziemlich schnell sehr zeitaufwendig, da bei jeder kleinen Änderung alle drei Source-Codes neu übersetzt werden müssen. Es ist besser, das Übersetzen und das Zusammenlinken zwei Schritten auszuführen. Um den `g++` anzuweisen, nur die ersten drei Schritte der Compilierung auszuführen, muß man die Option „-c“ übergeben:

```
g++ -c example.cc -o example.o
g++ -c classes1.cc -o classes1.o
g++ -c classes2.cc -o classes2.o
g++ example.o classes1.o classes2.o -o example
```

Wenn nun z.B. eine Änderung in classes1.cc vorgenommen wurde, so braucht nur der 2. und der 4. Befehl wiederholt zu werden. Die Entscheidung, welche Teile neukompiliert werden müssen, kann einfach anhand des jeweiligen Dateidatums gefällt werden. Ein Programm, das einem diese Arbeit abnimmt gibt es natürlich auch (siehe „make“ in Kapitel 3). Wichtig: Ein Source-Code muß natürlich auch dann neu kompiliert werden, wenn sich ein Header-File, daß mit `#include` eingefügt wird, ändert.

2.2.2 Einbinden von zusätzlichen Bibliotheken

Bei fast allen Programmen wird man auf weitere Bibliotheken (außer den Standard-Bibliotheken) zugreifen wollen. Dafür muß sowohl dem Präprozessor gesagt werden wo er die Header-Files dazu findet (wenn diese nicht an der Unix-Standard-Position `/usr/include` zu finden sind) und dem Linker muß man die Namen der Bibliotheken übergeben und evtl. auch das Verzeichnis, wenn diese Bibliothek nicht in `/lib` oder `/usr/lib` liegt. Beispiel 1: Nutzung der tiff-Bibliothek, die an der Standard-Position installiert ist:

```
g++ -c example.cc -o example.o
g++ -c classes1.cc -o classes1.o
g++ -c classes2.cc -o classes2.o
g++ example.o classes1.o classes2.o -o example -ltiff
```

Beispiel 2: Nutzung der FFTW-Bibliothek[3], die man im eigenen Home-Verzeichnis installiert hat (in /home/ronneber/include/fftw.h und /home/ronneber/lib/libfftw.a):

```
g++ -c -I/home/ronneber/include/ example.cc -o example.o
g++ -c -I/home/ronneber/include/ classes1.cc -o classes1.o
g++ -c -I/home/ronneber/include/ classes2.cc -o classes2.o
g++ -L/home/ronneber/lib example.o classes1.o classes2.o -o example -lfftw
```

Wenn man mehrere Bibliotheken einbindet, muß man unbedingt auf die richtige Reihenfolge achten. Jede Bibliothek kann nur die Funktionen der weiter hinten definierten nutzen. Beispiel: Die „fftw_threads“ - Bibliothek benötigt Funktionen aus der „pthread“ Bibliothek. Dann muß man beim Linken

```
-lfftw_threads -lpthreads
```

angeben. Nicht andersherum, dann findet der Linker die Funktionen nicht. Das dem so ist, ist übrigens Absicht. Andernfalls wäre es nicht möglich nur einige Funktionen aus einer Bibliothek mit einer eigenen Bibliothek zu überschreiben.

2.2.3 Erzeugen von eigenen Bibliotheken

Spätestens im Rahmen des Softwarepraktikums wird man mehrere Programme entwickeln, die alle auf dieselben Funktionen zugreifen. Diese sollte man rechtzeitig in eigene Bibliotheken legen, sonst können auch kleinste Änderungen schnell zur Tortur werden. Zum Erstellen einer Bibliothek benötigt man zusätzlich das Programm „ar“ (archive) und auf manchen Architekturen „ranlib“. Beispiel: Erstellen der Bibliothek „hurz“ mit allen Funktionen aus classes1.cc und classes2.cc:

```
g++ -c classes1.cc -o classes1.o
g++ -c classes2.cc -o classes2.o
ar -rcv libhurz.a classes1.o classes2.o
ranlib libhurz.a
```

Das Programm „ar“ hängt einfach die angegebenen Object-Files zusammen und erstellt einen Index, mit dem die Original-Files wieder hergestellt werden könnten. „ranlib“ extrahiert die Symbol-Tables der einzelnen Object-Files und faßt sie in einem großen Symbol-Table zusammen. Dies beschleunigt nachher das Linken, ist aber nicht unbedingt notwendig. Zum Einbinden dieser eigenen Bibliothek gilt dasselbe wie im vorigen Kapitel, um also unser example-Programm nun zu erstellen, schreiben wir nur noch

```
g++ -I/home/ronneber/hurz/include -c example.cc -o example.o
g++ -L/home/ronneber/hurz/lib example.o -o example -lhurz
```

[3]Sie enthält die „Fastest Fourier Transform in the West“ (`http://www.fftw.org`)

2.2.4 Weitere Optionen für den `g++`

Weitere nützliche Kommandozeilen-Optionen für den `g++` sind

- **-Wall** zeigt alle Warnungen an. Diese Compiler Warnungen deuten auf mögliche Programmierfehler hin, die aber nicht einen Abbruch der Compilierung erzwingen. Prädikat: sehr sinnvoll!
- **-O2** optimiert das erzeugte Programm auf Geschwindigkeit. Dadurch wird es üblicherweise etwas größer und das Compilieren dauert länger, aber gerade bei einfachen Schleifen (wie sie ja in der Bildverarbeitung dauernd vorkommen) bringt diese Optimierung einen enormen Geschwindigkeitsgewinn. Außerdem kann der Compiler beim Optimieren noch weitere Fehler finden, die dann als Warnungen ausgegeben werden, wie z.B. „variable xy might be used uninitialized“. Allerdings muß man sich beim Debuggen von optimierten Programmen auf einige Überraschungen gefaßt machen, da der gcc auch z.B. die Reihenfolge der Befehle umtauschen kann.
- **-g** schreibt die Informationen, die der Debugger benötigt in die Object-Files. Das sind vor allem eine Tabelle, die für jede Sourcecode-Zeile die Speicherposition des erzeugten Maschinen-Codes enthält, aber auch die Speicherposition und Namen von Variablen, etc.
- **-v** (verbose) schreibt während des Compilierens alle Zwischenschritte und die Verzeichnisse, in denen gesucht wird, heraus. Sehr sinnvoll, wenn man wissen will, woher die Bibliotheken nun eigentlich kamen...

2.2.5 Sonstige Utilities

Im Zusammenhang mit dem `g++` sollte auch noch kurz das Programm „strip“ erwähnt werden. Beispiel: mit

```
strip example
```

können nachträglich alle überflüssigen Symbol-Tables und Debugging-Informationen von einem Programm entfernt werden, wodurch es üblicherweise wesentlich kleiner wird. Man muß also nicht neu kompilieren, nur um ein Programm ohne Debugging-Informationen zu haben.

2.3 Referenzen

- `http://gnu.gcc.org` (Homepage des gcc)
- man gcc (Manual-Seite, gute tabellarische Auflistung der Optionen)
- info gcc (info-Seite, für die, die es ganz genau wissen wollen)

3 GNU `make`

Um den Quellcode eines Programms in einen ausführbaren Zustand — ausführbare Binärdatei — zu überführen sind verschiedenste Schritte notwendig. Zunächst einmal müssen alle Quelldateien des Programms vom Compiler in einen Objektcode überführt werden. Einige Objektdateien werden vielleicht zu Bibliotheken zusammengeführt. Schlußendlich müssen alle Objektdateien und Bibliotheken zu einer Binärdatei zusammengebunden (gelinkt) werden.

Diese Arbeitsschritte sind je nach Umfang des Softwarepaketes recht komplex und zeitaufwendig. Sicherlich ließen sich die Befehle statisch in ein Skript codieren. Doch würde der Zeitaufwand immer noch immens sein, wenn immer alle Quelldateien neu übersetzt werden müßten. Auch, wenn sich nur eine Quelldatei geändert hat. Sinnvoller und auch Ressourcen sparender wäre ein Werkzeug, das erkennt, welche Quelldateien neu compiliert, oder welche Bibliotheken neu gebildet werden müssen.

Genau dafür wurde das Werkzeug `make`[4] entwickelt. Mit dem Programm `make` ist es möglich, Abhängigkeiten zwischen beliebigen Dateien (z.B. Quelldateien und Objektdateien) zu definieren und Abbildungsvorschriften anzugeben, die einen Dateityp in einen anderen überführen (z.B. *.cc in *.o).

In diesem Kapitel wird eine kurze Einführung zu `make` gegeben. Es ist nur eine kurze Einführung, da das `make`-Werkzeug von so großer Komplexität ist, daß eine ausführliche Beschreibung den Rahmen dieser Anleitung sprengen würde.

3.1 Grundlagen

Im allgemeinen erwartet `make` eine Datei mit dem Namen `Makefile`, die im aktuellen Arbeitsverzeichnis steht. In der Datei `Makefile` ist definiert, was `make` machen soll. Beim Aufruf vom `make` liest das Programm das `Makefile` ein und interpretiert es. Soll `make` eine Datei mit einem anderen Name als `Makefile` nehmen, so läßt sich `make` mit dem Parameter `-f NameOfMakefile` aufrufen.

Damit `make` weiß, was es machen soll, sind in der `Makefile`-Datei Regeln definiert. Alle Regeln in einem `Makefile` müssen folgender Syntax genügen:

```
Ziel ... : Abhängigkeiten ...
<TAB>Befehl
<TAB>...
```

Demnach ist ein *Ziel* abhängig von seinen *Abhängigkeiten*. Wie das Ziel gebildet wird, steht in den Befehlszeilen darunter. Eine Regel kann auch nur aus *Ziel* und *Abhängigkeiten* bestehen. Ferner kann ein *Ziel* auch ohne *Abhängigkeiten* definiert sein, das nur Befehlszeilen beinhaltet.

[4] Diese Beschreibung von `make` basiert auf GNU `make` und lehnt sich an die Dokumentation von GNU `make` an. Achtung: Jedes Betriebssystem (z.B. Solaris, Irix, AIX) hat sein eigenes `make`, was selten vollkommen kompatibel zu GNU `make` ist. Um Ärger bei Portierungen von Software aus dem Weg zu gehen, sollte GNU `make` verwendet werden, da GNU `make` für alle gängigen Plattformen verfügbar ist.

Das Tabulatorzeichen zu Beginn jeder Befehlszeile ist obligatorisch. Fehlt das Zeichen, wird sich `make` bitterlich beschweren und die Ausführung des `Makefiles` abbrechen. Das Kommentarzeichen für ein `Makefile` ist das #-Zeichen.

Soll `make` nur ein *Ziel* erzeugen, muß `make` das Ziel in der Kommandozeile als Parameter mitgegeben werden. Ansonsten wird `make` versuchen, alle Regel zu befolgen.

`make` unterscheidet zwischen impliziten und expliziten Regeln. Implizite Regel sind Regeln, die `make` schon kennt und nicht definiert werden müssen. Zum Beispiel kann `make` eine C-Quelldatei `*.c` in eine Objektdatei `*.o` durch Compilation überführen. Explizite Regeln sind in dem `Makefile` definiert.

3.2 Ein einfaches `Makefile`

Für mehr Klarheit soll das folgende Beispiel sorgen. Ein Programm `myprg` soll aus drei Quelldateien `first.cc`, `second.cc` und `myprg.cc` gebildet werden. Das dazugehörige `Makefile` sieht wie folgt aus:

```
myprg :  myprg.o first.o second.o
        g++ myprg.o first.o second.o -o myprg

myprg.o : myprg.cc first.hh second.hh basics.hh
        g++ -c myprg.cc

first.o : first.cc first.hh
        g++ -c first.cc

second.o : second.cc second.hh
        g++ -c second.cc

clean :
        rm -f myprg myprg.o first.o second.o

rebuild : clean myprg
```

Nun soll `make` das Programm `myprg` erzeugen. Dafür wird `make` ohne Parameter aufgerufen. Das `make`-Programm liest das `Makefile` ein und interpretiert es. In Abhängigkeit existierender Dateien führt es die Regeln aus.

Zu Beginn gehen wir von der Existenz aller Quelldateien (*.cc und *.hh) aus. Die Objektdateien (*.o) und die Programmdatei `myprg` sollen noch nicht erzeugt worden sein.

In der ersten Regel des `Makefiles` ist definiert, wovon die Binärdatei `myprg` abhängt und wie das Programm gebildet wird. Demnach müssen für das Linken des Programms zunächst alle Objektdateien `myprg.o`, `second.o` und `first.o` generiert werden. Um das zu erfüllen sucht `make` jetzt nach Regeln für die Erzeugung der Objektdateien.

In der zweiten bis vierten Regel ist definiert, wie `make` die Objektdateien, die für das Ziel `myprg` notwendig sind, erzeugen kann. Zum Beispiel ist die Objektdatei `myprg.o` auf die

Quelldatei `myprg.cc` und den Headern `first.hh`, `second.hh` und `basics.hh` angewiesen. An dieser Stelle kann `make` die erste Regel erfüllen, da die Abhängigkeiten als Dateien existieren, und nicht erst gebildet werden müssen. D.h. `make` führt den Befehl `gcc -c myprg.cc -o myprg.o` aus und erzeugt so die Objektdatei `myprg.o`. Sukzessive werden so alle Objektdateien nacheinander — in der Reihenfolge wie es in der ersten Regel definiert ist — erzeugt.

`make` wird nun die Befehlszeile der ersten Regel ausführen, da die Abhängigkeiten erfüllt wurden. Mit der Befehlszeile werden alle Objektdateien zum Hauptprogramm `myprg.o` gelinkt und das ausführbare Programm `myprg` ist erzeugt.

Die vorletzte Regeln mit dem Ziel `clean` dient dem Aufräumen. Möchte man das Programm von Grund auf neu generieren, sorgt die Regel `clean` für das Löschen aller Objektdateien und der Programmdatei. Der Aufruf der Regel erfolgt mit `make clean`.

Um das Neuerzeugen noch weiter zu vereinfachen gibt es noch die Regel mit dem Ziel `rebuild`. Wird `make` mit dem Ziel `rebuild` aufgerufen, wird zunächst `clean` und dann `myprg` ausgeführt.

Wir nun zum Beispiel die Quelldatei `second.cc` geändert und anschließend der `make`-Befehl aufgerufen, so passiert folgendes: `make` liest wiederum das `Makefile` ein und schaut, ob alle Ziele erfüllt wurden. Bei der Regel mit dem Ziel `second.o` bemerkt das `make`-Programm, daß die Quelldatei `second.cc` ein aktuelleres Datum als die Objektdatei `second.o` hat. Ergo kann die Objektdatei nicht von der aktuellen Version der Quelldatei stammen. Somit führt `make` die zugehörige Befehlszeile aus und compiliert `second.cc` neu. Bis auf die erste Regel sind nun alle anderen Regeln erfüllt. Am Schluß bindet `make` alle Objektdateien zu einer neuen Version von `myprg` zusammen und terminiert. Es wurde nur die Arbeit vollzogen, die notwendig war.

3.3 `Makefiles` für Fortgeschrittene

Das zuvor diskutierte Beispiel für ein `Makefile` ist natürlich noch etwas unbeholfen. Und zwar deshalb, weil die Regeln für das Erzeugen der Objektdateien explizit für alle Dateien hingeschrieben wurde. Ferner wurden keinerlei Variablen verwendet. Soll z.B. der Compiler `gcc` ersetzt werden, oder ein zusätzlicher Parameter zum Aufruf zugefügt werden, müssen alle betreffenden Zeilen editiert werden.

Generell lassen sich Variablen in einem `Makefile` mit

```
VAR = WERT
```

definieren. Verwendet werden Variablen in einem `Makefile` mit

```
$(VAR).
```

Im folgenden Beispiel werden Compiler (`CXX`), Compilerparameter (`CXXFLAGS`), Linker (`CC`) und Objektdateien (`OBJS`) in Variablen definiert und verwendet. Ferner werden die impliziten Regeln von `make` für C++ Quellen (*.o:*.cc) verwendet:

```
OBJS = myprg.o first.o second.o
CXX = g++
CC = $(CXX)
CXXFLAGS = -Wall

all : myprg

myprg :  $(OBJS)

myprg.o first.o : first.hh

myprg.o second.o : second.hh

myprg.o : basics.hh

clean :
        rm -f myprg $(OBJS)

rebuild : clean myprg
```

Die Variablen `CXX`, `CC` und `CXXFLAGS` werden von den impliziten Regeln für C++ Übersetzung und Bindung verwendet. So teilt `CXXFLAGS` mit dem Flag `-Wall` dem Compiler mit, daß wirklich alle Warnungen ausgegeben werden sollen. Beim Aufruf von `make` wird, wie schon zuvor erklärt, versucht alle Regeln zu erfüllen. Wie `make` eine Objektdatei generiert ist in der impliziten Regel für `*.o:*.cc` erklärt. Um eine Quelldatei in eine Objektdatei abzubilden führt `make` den Befehl:

```
$(CXX) -c $(CPPFLAGS) $(CXXFLAGS)
```

aus. Die Abhängigkeiten der Objektdateien von den Headern steht nach wie vor in dem `Makefile`. Allerdings sind die Abhängigkeiten diesmal zusammengefaßt. Das Linken der Objektdateien wird über die folgende implizite Regel realisiert:

```
$(CC) $(LDFLAGS) Objektdateien $(LOADLIBES)
```

Die Verwendung von Variablen und impliziten Regeln führt somit zu wesentlich kompakteren `Makefiles`. Einen Katalog der impliziten Regeln findet man in der Anleitung zum GNU `make` (siehe Abschnitt Referenzen).

3.4 Referenzen

- Offizielle Homepage zu GNU `make`:
  `http://www.gnu.org/software/make/make.html`
- Manuals zu GNU `make`:
  `http://www.gnu.org/manual/make/index.html`
- Info-Seiten für GNU `make`: `info make`

4 Emacs

4.1 Einleitung

Emacs ist ein sehr leistungsfähiger und selbstdokumentierender Editor der aus einem Lisp-Interpreter mit Funktionen zur Textverarbeitung besteht. Emacs ist dadurch sehr flexibel an verschiedene Benutzeranforderungen anpaßbar und kann nahezu unbegrenzt, und ohne die Notwendigkeit einer Neukompilation, erweitert werden.

Ein wichtiges Grundkonzept von Emacs sind „Buffer“. Dies sind voneinander getrennte Arbeitsbereiche, in denen völlig unterschiedliche Dinge vor sich gehen können. Ein Buffer kann z.B. C++-Quellencode beinhalten, aber auch eine „Verzeichnisstruktur“, eine „Shell“, einen „Debugger“ oder einen „Compiler“.

Die Buffer sollten nicht mit den verschiedene „Frames“ (Fenstern) verwechselt werden. In den Frames werden ein oder mehrere Buffer angezeigt, was aber nicht bedeutet, daß ein Buffer der nicht angezeigt wird, nicht mehr existiert.

Ein weiteres wichtiges Grundkonzept sind die Modi, die es erlauben das Verhalten von Emacs an verschiedenen Bufferinhalte anzupassen. Im C++-Mode ist z.B. eine automatische Einrückung des Quellencodes mit `Tab` und die xfarblich Hervorhebung von Schlüsselwörtern möglich.

4.2 Erste Schritte

Vor dem Aufrufen muß in der Shell der Befehl `module add emacs` ausgeführt werden. Aufgerufen wird Emacs dann mit: `emacs filename`, wobei `filename` nicht mit angegeben werden muß.

Für Neubenutzer bietet das Tutorial, welches mit **C-h t** (bei gedrückter Control-Taste die h-Taste drücken und dann die t-Taste alleine drücken) aufgerufen wird, einen schnellen Einstieg.

4.3 Dokumentation

Zusätzlich zur im Tutorial beschriebenen Online-Hilfe (über die man sich mit **C-h ?** eine Übersicht verschaffen kann) können mit **C-h i** die Info-Files konsultiert werden. Das komplette Handbuch kann unter `http://www.gnu.org/manual/emacs-20.3/emacs.html` gefunden werden. Eine einseitige Zusammenfassung der wichtigsten Befehle (die Reference-Card) steht unter `http://www.refcards.com/` zur Verfügung.

4.4 Konfiguration von Emacs

Die Konfiguration von Emacs geschieht durch Editieren der Datei `.emacs` im Homeverzeichnis des Users, die beim Aufstarten von Emacs eingelesen wird.

Wenn zum Beispiel die Funktion *dabbrev-expand*, die praktisch ist um z.B. lange Variablenname nach Eingabe von ein paar Zeichen automatisch zu komplettieren, mit `S-iso-lefttab`

(gleichzeitiges Drücken der Shift- und der Tabulatortaste) anstatt des auf der deutschen Tastatur weniger praktischen `M-/` aufgerufen werden soll, kann dies durch Eintragen der folgenden Zeile in `.emacs` erreicht werden:

(global-set-key [S-iso-lefttab] 'dabbrev-expand)

4.5 Nützliches

Unbegrenztes Undo Mit `C-_` können alle Änderungen in einem Buffer rückgängig gemacht werden.

Inkrementale Suche Die nach `C-s` nach und nach eingegebenen Buchstaben werden bei der Suche berücksichtigt. Dadurch ist es möglich mit weniger Schreibaufwand Zeichenketten zu finden.

Kompilieren Ein C++-File kann mit `M-x compile` kompiliert werden. Falls Fehler auftreten kann durch drücken von `Return` (oder mit der mittleren Maustaste) in der entsprechenden Zeile des Kompilationsbuffers direkt an die Stelle des Fehlers im Quellcode gesprungen werden.

Debuggen GDB kann durch `M-x gdb` in Emacs aufgestartet werden und erlaubt eine komfortablere Bedienung als ein Aufruf in einer Shell.

Einfügen von Kommentaren Mit Hilfe des `abbrev-mode` können Abkürzungen definiert werden die nach der Eingabe in einem Buffer automatisch ersetzt werden. Somit ist es z.B. möglich das „Gerüst“ einer Funktionsbeschreibung einmalig zu erzeugen um es dann mit Hilfe der Abkürzung innerhalb des Quellcodes schnell einzufügen.

Navigieren im Quellcode Mit Hilfe des Programmes `etags` können für verschieden File-Typen tags (Markierungen) erzeugt werden. Diese tags können dann von Emacs gelesen werden und erlauben den schnellen Zugriff auf den Inhalt der Files (im Falle von C++-Files kann man z.B. so gezielt und schnell auf Funktionen zugreifen).

„Rectangle-Mode“ Das Kopieren, Ausschneiden, Einfügen und Füllen von rechteckigen Regionen ist mit dem „rectangle-mode“ möglich. Die Region kann mit der Maus ausgewählt werden (wobei Anfang- und Endpunkt die linke obere bzw. die rechte untere Ecke festlegen). Die wichtigsten Befehle sind:

- `C-x r k`: kill rectangle
- `C-x r y`: yank rectangle
- `C-x r t`: string rectangle

und können z.B. zum Auskommentieren verwendet werden.

5 GDB

5.1 Kurzbeschreibung

Ein Debugger ist ein Programm, das andere Programme ausführen kann und dem Benutzer Möglichkeiten zur Kontrolle und Untersuchung des Programmablaufs bietet. Insbesondere beinhalten diese Möglichkeiten kontrollierte schrittweise Abarbeitung des Programms und Anzeigen momentaner Variablen, Speicherzustände etc. Auch ist es möglich, aufgetretene Programmabstürze nachträglich zu untersuchen, und den Fehler so zu lokalisieren. Einige fortgeschrittenere Debugger bieten bequeme graphische Darstellung ganzer Datenstrukturen und deren Abhängigkeiten oder Performanceanalysen, um bei der Optimierung zu unterstützen.

Der GNU-Debugger GDB ist der populärste Debugger für UNIX-Systeme. Es ist ein Kommandozeilendebugger, der jedoch bequem durch Emacs oder andere grafische Benutzeroberflächen wie DDD bedient werden kann.

5.2 Wichtigste Anwendungsvarianten

5.2.1 Starten

Voraussetzung für das Debuggen eines Programms ist, daß das Kompilat entsprechende Informationen und Bezüge zu den Quelldateien enthält.

Dies muß beim Compilierenaufruf explizit angegeben werden durch das Flag `-g`. Folgender Aufruf erzeugt z.B. eine solche Datei.

```
g++ -g myfile.cc -o myfile
```

Auf ein solches Programm kann der GDB sofort angewandt werden. Gestartet wird er via `gdb`, notwendige Kommandos, um das Programm zu debuggen sind

- **file** *myfile*

  Dieser Befehl lädt die Debugging-Informationen aus der angegebenen Datei, und legt *myfile* als auszuführendes Programm fest. Dieser Befehl ist insbesondere nach Neucompilieren des Codes erforderlich!

- **run** *commandparams* [< *inputfile*] [> *outputfile*] Dieser Befehl startet das zuvor festgelegt Programm mit den Kommandozeilenparametern *commandparams* und den angegebenen Dateien als Quellen für stdin/stdout.

Weitere allgemeine Kommandos sind z.B.

- **help**

  Zeigt Hilfethemen an, bzw. durch Angabe eines konkreten Befehls dessen spezifische Hilfekommentare.

- **shell** *cmdstring*

  Führt den angegebenen String in der zugrundeliegenden Shell aus.

- **quit**

  Beendet den gdb.

5.2.2 Kontrolle des Programmablaufs

Die bisherigen Befehle bieten nicht mehr Funktionalität als ein direkter Aufruf des Programms aus einer Shell heraus. Wesentliche Elemente zur Steuerung des Programmablaufs sind nun die Befehle zum Anhalten der Ausführung an definierten Stellen (Breakpoints) und zum schrittweisen Ausführen:

- **break** [*file:*]*function*, **break** [*file:*]*line*

  Setzt einen Haltepunkt beim Eintritt in die angegebene Funktion, bzw. an der angegebenen Zeile im Quellcode.

- **info break**

  Listet alle bisher definierten Haltpunkte auf.

- **delete** [n]

  Löschen aller Haltepunkte oder des definierten Haltepunkts mit Nummer n.

- **step**, **s**

  Ausführen der aktuellen Zeile des Programmcodes. Ist dies ein Aufruf einer Funktion, wird an den Anfang von deren Quellcode gesprungen.

- **next**, **n**

  Ausführen der aktuellen Zeile des Programmcodes. Ist dies ein Aufruf einer Funktion, wird die gesamte Funktion abgearbeitet, und nicht in diese Funktion hineingesprungen.

- **up**, **down**

  Springt in der Funktionsaufruf-Hierarchie eine Stufe nach oben bzw. unten, ermöglicht hiermit Untersuchung der Funktionsaufrufe und deren Parameter, die zu einer bestimmten Programmstelle geführt haben.

- **finish**

  Führt das Programm bis zum Ende der aktuellen Funktion aus.

- **cont**, **c**

  Führt das Programm normal aus, bis der nächste Haltepunkt erreicht wird, oder das Programm terminiert.

5.2.3 Speicherkontrolle

Weitere wesentliche Funktionalität eines Debuggers umfaßt Zugriffsmöglichkeiten auf den Speicher, d.h. Speicher auslesen bzw. Speicherinhalte manipulieren:

- **print** *expr*, **p** *expr*

  Wertet den C-Ausdruck *expr* (normalerweise eine Variable) einmalig aus, und stellt das Ergebnis dar.

- **display** *expr*

  Wertet den C-Ausdruck *expr* aus, und stellt das Ergebnis bei jedem folgenden Programmstop dar.

- **undisplay** *n*

  Löscht den anzuzeigenden Ausdruck mit Nummer *n* von der Liste der wiederholt anzuzeigenden Ausdrücke.

- **set** *var* = *expr*

  Wertet den Ausdruck *expr* aus, und setzt dies als neuen Wert der Variablen *var*.

5.2.4 Untersuchung von Programmabstürzen

Außer kontrolliertem Ausführen von Programmen ermöglicht GDB auch die Untersuchung vorangehender Programmabstürze auf Basis eines beim Absturz erzeugten Corefiles. Die Erzeugung solcher Files muß in der Shell zunächst eingestellt werden. Mit Hilfe von

```
ulimit -a
```

werden die Speichergrenzen für diverse Größen angezeigt. Unter `coredump` muss ein positiver Eintrag stehen, damit Coredumps erzeugt werden. Dies kann (in den meisten Shells) via

```
ulimit -c coresize
```

eingestellt werden, wobei `coresize` die gewünschte Größe in Bytes darstellt.

Hat ein Fehlerhaftes Programm anschließend ein Corefile erzeugt, kann die Absturzstelle mit GDB ermittelt werden, indem nach normalem Starten von GDB und Festlegen des zu inspizierenden (abgestürzten) Programms mit **file**, das Corefile geladen wird. hierzu dient der Befehl

- **core** *corefile*

Der Programmablauf befindet sich exakt am Absturzpunkt. Ein Abarbeiten des Programms ist selbstverständlich nicht weiter möglich, jedoch lassen sich die Variablen weiter abfragen.

5.3 GDB unter Emacs

Speziell unter Verwendung von Emacs läßt sich der GDB bequem bedienen. Mit **M-x gdb** wird ein Fenster geöffnet, in dem GDB-Kommandos ausgeführt werden können. Indem man im Hauptprogramm einen Haltepunkt setzt und das Programm startet, teilt sich das aktuelle Fenster, und der Quellcode erscheint. Emacs interpretiert die Informationen von gdb, indem es die aktuelle Position im Sourcefile markiert, und dies auch beim schrittweisen Abarbeiten des Programms immer aktualisiert.

Spezielle Tastenkombinationen ermöglichen einfachere Eingabe von Befehlen. u.a.

- **C-c C-c** Bricht Programmablauf ab, springt an die aktuelle Stelle im Programm und ermöglicht anschließend schrittweise Abarbeitung.
- **C-x SPC**

  Fügt einen Haltepunkt in der Zeile der aktuellen Cursorposition ein.
- **M-p, M-n**

  Die Liste der bisher eingegebenen GDB-Kommandos wird abgelaufen, ermöglicht so einfaches Wiederholen von Befehlen.

5.4 Referenzen

Die angegebenen Varianten der Befehle sind nur ein Bruchteil aller möglichen. Für ausführlichere Beschreibungen siehe folgende Referenzen:

- Online-Handbuch: `http://sources.redhat.com/gdb/onlinedocs/gdb_toc.html`
- Referenz-Karte: `http://www.cs.princeton.edu/~benjasik/gdb/gdb.ps`

6 CVS

6.1 Was ist CVS?

Das Concurrent Versions System (CVS) dient zur Versionsverwaltung kompletter und ggf. mehrstufiger Datei-Verzeichnisse, wie sie üblicherweise in größeren Software- oder Konfigurationsprojekten entstehen. CVS baut intern auf das sogenannte Revision Control System (RCS) auf und benutzt eine ähnliche Begriffswelt wie RCS. Im Gegensatz zu RCS sind jedoch keine expliziten Sperren auf Dateien möglich, so daß auch mehrere Personen die gleiche Datei ändern (oder gar löschen) können. Etwaige Konflikte, die bei den meisten Projekten erfahrungsgemäß jedoch selten auftreten, werden später aufgelöst.

Die Grundidee bei CVS ist, daß es ein zentrales Repository (auf deutsch: Lagerstätte) gibt, in dem der gesamte Dateibaum verschiedener Projekte mit den dazugehörigen Verzeichnissen, Dateien und deren Versionen abgelegt ist. Jeder Benutzer erzeugt mit Hilfe von CVS eine private Arbeitskopie (oder auch mehrere) eines zu einem Projekt gehörenden

Dateibaums. Hier kann er die Dateien beliebig ändern, sowie neue Dateien und Verzeichnisse erzeugen oder auch veraltete Dateien und Verzeichnisse entfernen. Nachdem er seine Änderungen vorgenommen hat, muß er diese zunächst mit der Version im Repository abgleichen und anschließend in das Repository eintragen, nachdem er ggf. Konflikte mit den Änderungen anderer Benutzer aufgelöst hat. Hilfreich ist hier oft, sich zunächst die eigenen Änderungen nochmals anzusehen, insbesondere, wenn mehrere Dateien betroffen sind.

Anmerkungen: CVS wird fast vollständig durch den Befehl cvs(1) gesteuert. Die eigentlichen Aktionen von CVS werden durch Kommandonamen gesteuert, die als erster Parameter auf der Kommandozeile dem Befehlswort cvs folgen (s.u.). An verschiedenen Stellen werden Versionsnummern und Bezeichner benötigt und benutzt, auf die wir hier aber nicht weiter eingehen.

6.2 Arbeiten mit CVS

6.2.1 Vorbereitungen

module add cvs — Programm in den Suchpfad laden Das Programm `cvs` muß der Shell bekannt gemacht werden. Das geschieht in unserer Umgebung durch die Eingabe von

```
idefix:~ $ module add cvs
```

cvs init — Neues CVS-Repository anlegen Ein neues Repository wird angelegt, indem man (an beliebiger Stelle) ein Verzeichnis anlegt, die Environment-Variable `CVSROOT` darauf einstellt, in dieses wechselt und dann das Kommando `init` ausführt:

```
idefix:~ $ mkdir /sw-praktikum/cvsroot
idefix:~ $ cd /sw-praktikum/cvsroot
idefix:~ $ setenv CVSROOT /sw-praktikum/cvsroot
idefix:/sw-praktikum/cvsroot $ cvs init
```

cvs import — Neues CVS-Projekt anlegen Man beginnt ein neues Projekt, indem man in einem beliebigen Verzeichnis `work_dir` (nicht in `CVSROOT`!) mindestens ein neues Verzeichnis anlegt, ggf. auch weitere Unterverzeichnisse des neuen Projekts. Anschließend wechselt man wieder nach `work_dir` und ruft dort das CVS-Kommando import auf. Dabei werden drei weitere Parameter nötig:

1. Der Projektname (Verzeichnisname), unter dem das Projekt abgelegt werden soll.

2. Ein globaler Bezeichner für die Version, der vor allem wichtig ist, wenn mehrere Versionen von Fremdsoftware mit verwaltet werden sollen (hier also nicht).

3. Ein lokaler Bezeichner für die Version.

Die lokalen bzw. globalen Bezeichner sind für uns in diesem Falle (neues Projekt) nicht weiter von Interesse. Wir wählen einfache Bezeichner, die auf die Neuheit des Projektes verweisen:

```
idefix:...  $ mkdir my_proj
idefix:...  $ cd my_proj
idefix:.../my_proj $ mkdir src
idefix:.../my_proj $ mkdir include
idefix:.../my_proj $ cvs import new_proj start_it developer_0
cvs import:  Importing $CVSROOT/new_proj/src
cvs import:  Importing $CVSROOT/new_proj/include
    -->$CVSEDITOR
No conflicts created by this import
```

6.2.2 Arbeiten mit einem CVS-Repository

cvs checkout — Arbeitskopie erzeugen Nachdem man die Environment-Variable `$CVSROOT` auf den Pfad des Repositories eingestellt hat, kann man mit dem CVS-Kommando `checkout` im aktuellen Verzeichnis eine Kopie der aktuellen Version des Dateibaums von einem im Repository abgelegten Projekt erzeugen (hier z.B. test):

```
idefix:~/...  $ cvs checkout test
cvs checkout:  Updating test
U test/file
```

Nun können die bestehenden Dateien geändert (bzw. neue angelegt oder veraltete gelöscht) werden:

```
idefix:~/...  $ cd test
idefix:~/.../test $ $EDITOR file
```

Anmerkung: CVS legt im Arbeitsverzeichnis (und in allen Unterverzeichnissen) zusätzliche Verzeichnisse mit dem Namen CVS an. Diese und die darin enthaltenen Dateien dürfen nicht manipuliert werden!

cvs update — Eigene Änderungen mit dem Repository abgleichen Nach erfolgter Änderung werden die Dateien mit dem CVS-Kommando `update` mit der Version im Repository verglichen. Hierbei wird insbesondere geprüft, ob seit dem letzten Abgleich (oder checkout) Dateien durch andere Benutzer geändert wurden. Existieren solche Änderungen, werden sie automatisch in der eigenen Arbeitskopie nachgetragen. Falls dabei Konflikte auftreten, z.B. weil zwei Benutzer an der gleichen Stelle einer Datei geändert haben, so müssen diese zunächst aufgelöst werden.

```
idefix:~/.../test $ cvs update
cvs update:  Updating .
M file
```

Es wird für jede Datei, die von Änderungen betroffen ist, ihr Name ausgegeben und ein Indikator (Buchstabe am Anfang der Zeile), der die Änderung beschreibt:

Updated Änderungen aus dem Repository wurden in die Arbeitsdatei übernommen. Es sind keine Konflikte aufgetreten.

Modified Die Arbeitsdatei wurde verändert. Änderungen können durch ein `commit` ins Repository übertragen werden.

Conflicts Es gibt Konflikte zwischen Änderungen in der Arbeitsdatei und Änderungen im Repository (s. Konflikte auflösen).

Removed Die Arbeitsdatei wird (durch das `commit`) auch im Repository gelöscht.

Added Die Arbeitsdatei wird (durch das commit) als neue Datei ins Repository aufgenommen.

? Arbeitsdatei: Keine korrespondierende Datei im Repository

cvs commit — Änderungen ins Repository eintragen Treten keine Konflikte auf, bzw. sind diese behoben, so werden die Änderungen endgültig mit dem CVS-Kommando `commit` ins Repository eingetragen. Dabei wird für die geänderten Dateien ein Editor (Environment-Variable `$CVSEDITOR` bzw. `$EDITOR`) geöffnet, in den man seine Anmerkungen zu den Änderungen eintragen kann:

```
idefix:~/.../test $ cvs commit
cvs commit:  Examining .
cvs commit:  Committing .
```

```
    -->$CVSEDITOR
Checking in file;
$CVSROOT/test/file,v  <--  file
new revision:  1.2; previous revision:  1.1
done
```

Achtung: Wurden die Konflikte nach einem `update`-Kommando nicht aufgelöst, so werden die Dateien mit Konfliktmarkierungen (s. Konflikte auflösen) ins Repository übertragen. Das Arbeitsverzeichnis bleibt durch den Befehl commit erhalten, es kann einfach gelöscht werden, falls es nicht mehr gebraucht wird.

cvs add — Neue Dateien einfügen Hat man, z.B. mit dem Editor, eine neue Datei angelegt, so muß diese mit dem CVS-Kommando `add` beim Repository angemeldet werden.

```
idefix:~/.../test $ $EDITOR newfile
idefix:~/...;/test $ cvs add newfile
cvs add:  scheduling file ‘newfile’ for addition
cvs add:  use ’cvs commit’ to add this file permanently
```

Die endgültige Übernahme erfolgt erst bei einem `commit`:

```
idefix:~/.../test $ cvs commit
cvs commit:  Examining .
cvs commit:  Committing .
RCS file:  $CVSROOT/test/newfile,v
done
    -->$CVSEDITOR
Checking in newfile;
$CVSROOT/test/newfile,v  <--  newfile
initial revision:  1.1
done
```

cvs remove — Dateien löschen Dateien bzw. Verzeichnisse können durch das CVS-Kommando `remove` aus dem Repository gelöscht werden, wenn sie auch im Arbeits-

verzeichnis gelöscht wurden (bei Verzeichnissen erst deren Dateien aus dem Repository löschen, ggf. rekursiv).

```
idefix:~/.../test $ rm file
idefix:~/.../test $ cvs remove file
cvs remove:  scheduling ‘file’ for removal
cvs remove:  use ’cvs commit’ to remove this file permanently
```

Auch hier erfolgt die endgültige Löschung erst durch ein `commit` (Die Dateien werden nicht wirklich im Repository gelöscht, sondern nur als gelöscht markiert, da man jederzeit eine alte Version des Projektes wieder herstellen können muß):

```
idefix:~/.../test $ cvs commit
cvs commit:  Examining .
cvs commit:  Committing .
    -->$CVSEDITOR
Removing file;
$CVSROOT/test/file,v  <--  file
new revision:  delete; previous revision:  1.2
done
```

cvs diff — Änderungen betrachten Oft will sich der Benutzer vor einem `commit` nochmals die Änderungen ansehen, die er vorgenommen hat. Hier gibt es das CVS-Kommando `diff`, das auf dem UNIX-Befehl `diff(1)` aufbaut und auch die gleichen Optionen hat. In diesem Fall werden Änderungen mit einem `!` eingeleitet:

```
idefix:~/.../test $ cvs diff -bc
cvs diff:  Diffing .
Index:  file
===================================================================
RCS file:  $CVSROOT/test/file,v
retrieving revision 1.1
diff -b -c -r1.1 file
*** file        1998/01/20 00:03:59     1.1
```

```
--- file        1998/01/21 08:15:21
***************
*** 3,9 ****
  #
  # test CVS -- history at end

!  testfile

  #
  # Revision 1.4  2000/01/13 08:25:19  duda
  # Seitenfuss vereinheitlicht
  #
  # Revision 1.3  2000/01/12 09:56:11  duda
  # ISA-->VS
  #
  # Revision 1.2  1998/07/01 09:35:28  ascheman
  # Rück-Verweis auf CVS-Hauptseite eingeführt
  #
  # Revision 1.1  1998/04/20 11:13:39  ascheman
  # Initial revision
  #
--- 3,10 ----
  #
  # test CVS -- history at end

!  changed one line
!  to two lines!

  #
  # Revision 1.4  2000/01/13 08:25:19  duda
  # Seitenfuss vereinheitlicht
  #
  # Revision 1.3  2000/01/12 09:56:11  duda
```

```
# ISA-->VS
#
# Revision 1.2  1998/07/01 09:35:28  ascheman
# Rück-Verweis auf CVS-Hauptseite eingeführt
#
# Revision 1.1  1998/04/20 11:13:39  ascheman
# Initial revision
#
```

Konflikte auflösen Treten bei dem Kommando `update` Konflikte auf, z.B.

```
idefix:~/.../test $ cvs update
cvs update:  Updating .
RCS file:  $CVSROOT/test/newfile,v
retrieving revision 1.1
retrieving revision 1.2
Merging differences between 1.1 and 1.2 into newfile
rcsmerge:  warning:  conflicts during merge
cvs update:  conflicts found in newfile
C newfile
```

die nicht von CVS automatisch aufgelöst werden können (Dateien, die mit dem Indikator `C` gekennzeichnet sind), so passiert folgendes:

1. Die unmodifizierte Datei, von der beide Versionen ausgegangen sind, wird als *.#file.version* im jeweiligen Arbeitsverzeichnis abgelegt. Dabei entspricht *file* dem Dateinamen und *version* der Versionsnummer.
2. Die eigentliche Datei enthält den kompletten Inhalt beider Versionen inkl. der Konflikte. Bereiche, in denen Konflikte aufgetreten sind, sind folgendermaßen aufgebaut:

   ```
   <<<<<<< newfile
   Conflict from local file!
   =======
   Conflict from repository!
   ```

```
>>>>>>> 1.2
```

Die Konflikte müssen dann in der Arbeitsdatei manuell aufgelöst werden. Anschließend muß die Datei (oder die Dateien) wieder mit den Befehlen update und commit ins Repository übertragen werden.

6.2.3 CVS und Emacs

Die meisten CVS Kommandos lassen sich komfortabel im Emacs ausführen.

Als Vorbereitung muß dafür (vor dem Start des emacs) in der Shell der Befehl

```
module add emacs
```

ausgeführt werden. Folgende Zeile ist außerdem in die Datei `~/.emacs` zu übernehmen:

```
(autoload 'cvs-update "pcl-cvs" nil t)
```

Der CVS-Modus kann dann durch den Emacs-Befehl

```
M-x cvs-update
```

aktiviert werden. Hilfe zu den verfügbaren Kommandos gibt anschließend die Eingabe eines `?`.

Alternativ dazu kann auch der Menüpunkt Tools→Version Control verwendet werden.

6.2.4 CVS @ Home

CVS kann auch von einem Heimarbeitsplatzrechner verwendet werden. Zunächst muss man gewährleisten, dass eine ordnungsgemäße `ssh`-Verbindung vom Heimarbeitsplatz zu dem CVS-Server aufgebaut werden kann. Näheres dazu unter `man ssh, man ssh-keygen`.

Weiterhin müssen folgende Variablen (auf dem Heimarbeitsplatz) gesetzt werden:

```
heimrechner:~ $ setenv CVSROOT USERNAME@SERVERNAME:/sw-praktikum/cvsroot
heimrechner:~ $ setenv CVS_RSH ssh
```

`USERNAME` steht für den Uni-Account, als `SERVERNAME` kann `miraculix.informatik.uni-freiburg.de` eingesetzt werden. Anschließend kann wie gewohnt durch den Befehl

```
heimrechner:~ $ cvs checkout test
```

eine Arbeitskopie erzeugt werden.

6.3 Referenzen

Die hier beschriebene Kurzanleitung orientiert sich zu einem großen Teil an
`http://www.informatik.tu-darmstadt.de/VS/Rechner/Software/Cvs/Kurz.html`

Weitere Referenzen lassen sich ausgehend von `http://www.cvshome.org` lesen.

Weiterhin stehen im Labor die Handbücher „Version Management with CVS“, „User's Guide to pcl-cvs—the Emacs Front-End to CVS“ und „CVS Quick Reference Card“ bereit.

7 Doxygen

7.1 Kurzbeschreibung

Doxygen ist ein Dokumentationssystem für C++, Java, IDL und C. Es kann auf 3 verschiedene Weisen eingesetzt werden:

1. Es kann auf Basis von kommentierten Quelldateien on-line Dokumentationen in HTML und/oder off-line Handbücher in LaTeX erzeugen. Ausgabe in RTF (MS-Word), PostScript, hyperlinked PDF, komprimiertem HTML und als Unix man pages werden ebenfalls unterstützt. Die Dokumentation wird direkt aus den Quelldateien extrahiert, wodurch die Konsistenz von Programmcode und Dokumentation sehr viel einfacher gewährleistet werden kann.

2. Doxygen kann dazu verwendet werden die Programmstruktur aus unkommentierten Sourcen zu extrahieren. Dies kann sehr nützlich sein, um sich schnell in großen Paketen von Quelldateien zu orientieren. Die Abhängigkeiten zwischen Dateien, die hierarchischen Beziehungen zwischen Klassen und die Komponenten von Klassen bzw. Strukturen werden durch Graphen visualisiert, die alle automatisch generiert werden.

3. Doxygen kann durch seinen Befehlssatz (einschließlich einiger LaTeX und HTML-Kommandos) dazu mißbraucht werden, gewöhnliche Dokumente zu erzeugen.

7.2 Wichtigste Anwendungsvarianten

7.2.1 Setup und einfacher Aufruf

Um Doxygen benutzen zu können müssen zunächst entsprechende Softwarepakete eingebunden werden. Dies geschieht durch die shell-Kommandos

```
module add doxygen
module add gv
```

Die gesamten Optionen zur Funktionsweise von Doxygen müssen in einem Konfigurationsfile angegeben werden, das ähnliche Struktur wie ein Makefile hat.

Ein solches Default-Konfigurationsfile wird erzeugt, indem Doxygen mit der -g Option aufgerufen wird:

```
doxygen -g <conffile>
```

Anschließend kann die Datei `<conffile>` mit einem beliebigen Editor bearbeitet werden, oder man verwendet die grafische Oberfläche `doxywizard`, die über nützliche Hilfekommentaren zu den einzelnen Einträgen verfügt.

Die minimal anzugebenden Optionen sind

- `INPUT,FILEPATTERNS`: die zu durchsuchenden Dateien festlegen
- `HAVE_DOT`: Erzeugen der diversen Graphen
- `GENERATE_HTML,GENERATE_RTF,GENERATE_LATEX`, `GENERATE_XML,GENERATE_MAN`: Festlegen der Ausgabeformate.

Ohne weitere Angaben wird für die entsprechenden Ausgabeformate jeweils ein default-Verzeichnis angelegt, in das die Ausgabe erfolgt.

Weitere nützliche Optionen sind

- `OUTPUT_LANGUAGE`: Ausgabesprache festlegen
- `HTML_HEADER,HTML_FOOTER`: Festlegen der HTML-Files, die in der HTML-Dokumentation als Kopf bzw. Fuß jeder Seite eingefügt werden.

7.2.2 Kommentieren der Sourcen

Für sinnvolle Aussagen aus der durch Doxygen erzeugten Dokumentation ist ein vorgeschriebener Dokumentationsstil der Quelldateien notwendig. Durch einen Kommentarblock der Form `/*! text */` bzw. die Einzelzeilenvariante `//! text` können Kommentarblöcke definiert werden, die von Doxygen bearbeitet werden. Innerhalb dieser Blöcke kann beliebiger freier Text und eine Vielzahl von Schlüsselwörter verwendet werden.

Für alle Bestandteile wie Dateien, Klassen, Methoden, Variablen, Funktionen etc. ist eine solche Dokumentation möglich und sinnvoll. Grundsätzlich ist für jedes dieser Objekte eine Kurzdokumentation (durch die Einzeilenvariante) und eine detailliertere Dokumentation (durch einen Kommentarblock) möglich.

Ist eine Kurzdokumentation länger als eine Zeile, muß diese innerhalb des Detaildokumentationsblocks durch das Kommando `\brief` definiert werden. Der gesamte Text bis zur nächsten Leerzeile wird dabei als Kurzdokumentation aufgefaßt.

Die Kommentarblöcke stehen grundsätzlich vor den Definitionen oder Deklarationen. In Einzelfällen ist eine andere Position sinnvoll. Eine Referenz auf das *vorhergehenden* Objekt kann z.B. durch die Einzeilenvariante via `//!< text` hergestellt werden. Dies macht bei Variablen Sinn, die nur einer kurzen Erläuterung bedürfen, ein Beispiel folgt unten.

Will man einen Kommentarblock vollkommen von der Definition oder der Deklaration eines Objektes lösen, muß der Bezug des Blocks zu dem Objekt hergestellt werden mittels dem entsprechenden Schlüsselwort `\class, \struct, \union, \enum, \fn, \var, \def, \file` oder `\namespace` und anschließendem Objektnamen.

Grundsätzlich sollen `public`-Komponenten in den Header-Dateien dokumentiert werden, `private`-Komponenten stattdessen in den zugehörigen Quelldateien.

Einige Schlüsselwörter, die in den Blöcken verwendet werden sollen, sind:

- `\param <parname> <description>`: Dokumentation von Funktionsparametern.
- `\exception <exname> <description>`: Dokumentation möglicher Exceptions.
- `\return <description>` : Dokumentation des Rückgabewertes einer Funktion.

Nun wird ein Beispiel für eine solche kommentierte Datei angegeben. Man beachte, daß die 'Kommentarsternchen' von Doxygen sinnvollerweise nicht als Textbestandteile übernommen werden.

```
/*! ***************************************************************
**
** \file dummyclass.hh
** \brief provides dummy class for demonstration of
**        doxygen-functionality.
**
*****************************************************************/

#include <parentclass.hh>

  //! brief description of Dummyclass.
  /*! More detailed description can be
      longer. */

class DummyClass : public ParentClass
{
public:

  /*! \brief    a brief description can also last over several lines
   *            as shown in this example of a constructor.
   *
   *  detailed description of the constructor:
   *   creates new Dummyclass Instance.
   *
   *   \param      param1    some initialization parameter
   *   \param      name      name of the instance.
   *
   *   \exception  DummyClassException the possible exceptions
   */

    DummyClass(const int param1, const char *name);

  //!  calculation of some reasonable value.
  /*!
   *   \param      param1    some necessary parameter for calculation
   *
```

```
   *   \return     the result of the calculation
   */

    int calc_something(const int param1);

  /*!
      a public variable containing one property.
   */

    int property1;
    float property2; //!< this is a backward-reference documentation line.
};
```

7.2.3 Weitere Funktionalität

Die bisher angegebenen Schlüsselwörter sind nur ein geringer Ausschnitt aller möglichen. Unter anderem sind viele HTML und LaTeX Kommandos möglich, für Details hierzu und zu den vielfältigen Einträgen in den Konfigurationsfiles siehe unterstehende Referenz.

7.3 Referenzen

- Doxygen Homepage: `http://www.stack.nl/~dimitri/doxygen/index.html`