

Programmieren mit *Cygwin* und *UNIX*

Prof. Dr. Manfred Brill

Oktober 2003

Inhaltsverzeichnis

1	Ein kleines Beispiel zur Einführung	1
2	UNIX-Shells	2
3	Programmentwicklung in <i>UNIX</i> und <i>CYGWIN</i>	4
3.1	Edit	5
3.2	Compile	5
3.3	Make	6
3.4	Debug	11
4	Ausblick	12
	Literatur	12

1 Ein kleines Beispiel zur Einführung

Nehmen wir an, dass Sie ein kleines Programm geschrieben haben, beispielsweise das omnipräsente „Hello World!“. Der Quelltext könnte dann so aussehen:

```
#include <iostream>
void main(void)
{
    cout << "Hello out there!\n";
}
```

Weiter nehmen wir an, dass Sie *Cygwin* auf Ihrem Wintel-Rechner installiert haben. Öffnen sie eine *Cygwin*-Shell mit dem entsprechenden Symbol auf der Startleiste (suchen Sie nach dem Eintrag „Cygwin Bash Shell“). Wechseln Sie in das Verzeichnis, in dem Sie die Datei *hello.cpp* gespeichert haben und geben dort das Kommando *make hello* ein. Sie werden einige Ausgaben erhalten wie in Abbildung 1. Ist das Kommando abgeschlossen, dann starten Sie Ihr Programm

```
brill@STING[~/pcdata/Lectures/Xtra-texte/CC] >vi hello.cpp
brill@STING[~/pcdata/Lectures/Xtra-texte/CC] >make hello
g++    hello.cpp    -o hello
brill@STING[~/pcdata/Lectures/Xtra-texte/CC] >hello
Hello out there!
brill@STING[~/pcdata/Lectures/Xtra-texte/CC] >
```

Abbildung 1: Die Erstellung des Programms und die Ausgabe

durch Eingabe von *hello*. So einfach ist das! Vergleichen Sie dies mit der Beschreibung, wie man in *Microsoft Visual* oder in *Eclipse* das gleiche Programm zum laufen bringt. Das ging auf jeden

Fall schneller, oder? Warum das so funktioniert, und was man bei komplexeren Programmen berücksichtigen muss, das ist das Thema dieses Textes.

Sollten Sie bereits mit Shells wie der *Eingabeaufforderung* oder mit einem Terminalfenster in *LINUX* gearbeitet haben, dann können Sie den Rest dieses Abschnitts überspringen und gleich zu Abschnitt 3 übergehen!

2 UNIX-Shells

Wenn Sie Cygwin erfolgreich installiert haben, dann gibt es neben der *Eingabeaufforderung* eine weitere *Shell* auf Ihrem Rechner. Eine Shell dient verkürzt formuliert dazu, mit dem Betriebssystem zu interagieren, wie Sie das von *command.com* von *Windows* kennen.

Die Default-Shell von Cygwin und bei allen *LINUX*-Distributionen, die ich kenne ist die *bash* - die „Bourne Again Shell“. Dieser Name wurde von der ersten UNIX-Shell, die es überhaupt gab, der „Bourne Shell“ abgeleitet. Diese Shell ist extrem schnell, aber ebenso unhandlich. Deshalb wurde von *SUN Microsystems* die „C-Shell“ entwickelt, deren Syntax sich, wie der Name schon verspricht, an der Programmiersprache *C*, orientiert. Diese Shell wurde weiterentwickelt zur „TC-Shell“, die beispielsweise Funktionen wie History mit Hilfe der Cursortasten oder *Word Completion* bietet. Es gibt darüberhinaus noch eine „Korn Shell“ und eben die *bash*, die die Funktionalität der C-Shell und die Geschwindigkeit der Bourne Shell verbinden soll. Welche Shell man verwendet ist Geschmackssache und wird unter Programmieren und UNIX-Gurus sicher genauso heftig diskutiert wie das Für und Wider von Programmiersprachen. Als Default wird bei *Cygwin* auf jeden Fall die *bash* installiert, die anderen Shells können Sie jedoch auch installieren, wenn Sie möchten. In Tabelle 1 finden Sie die Befehle zum Start der erwähnten Shells. Sie können von einer Shell in eine andere wechseln, in dem Sie sie wie einen anderen Befehl im Cygwin-Fenster eingeben. Typischerweise finden Sie diese Dateien im Ordner */bin* oder */usr/bin*. Sie können in einer Shell eine weitere Shell starten; durch das Kommando *exit* beenden Sie eine Shell.

Bezeichnung	Aufruf
Bourne-Shell	sh
C-Shell	csh
TC-Shell	tcsh
Korn-Shell	ksh
Bourne Again Shell	bash

Tabelle 1: UNIX-Shells und ihr Aufruf

Die meisten Shells haben eine sogenannte *History*, damit ist eine Liste der von Ihnen eingegebenen Kommandos gemeint. Sie können mit den Cursortasten in dieser Liste scrollen. Den letzten Befehl können Sie darüberhinaus mit dem Kommando *!!* abrufen. Das Kommando *!-2* ruft das vorletzte Kommando auf und so weiter. Sie können auch auf die Argumente der letzten Kommandos zugreifen, was Ihnen viel Schreibarbeit sparen kann. Mit *!-1:1* greifen Sie auf das erste Argument des letzten Kommandos zu; *!-1:0* ist identisch mit *!-1* selbst. Mit dem Kommando *history* erhalten Sie eine Liste Ihrer Kommandos; wie lange diese ist wird in einer Systemeinstellung festgelegt.

Bei der Angabe des Ordners, in dem Sie die Shell-Kommandos typischerweise finden haben Sie bestimmt bemerkt, dass die Verzeichnisnamen in UNIX-Systemen mit */* getrennt werden. Darüberhinaus ist in UNIX die Groß- und Kleinschreibung immer signifikant. Die Datei *Hello* ist also eine Andere wie die Datei *hello* – das ist häufig eine beliebte Fehlerquelle beim Umstieg von *Windows* auf *UNIX*. Wie in *Windows* gibt es auch in *UNIX* die Möglichkeit, *hidden files* anzulegen, die bei der „normalen“ Auflistung eines Ordnerinhalts nicht angezeigt werden. Beginnt der Dateiname mit einem „.“, dann ist damit eine versteckte Datei gekennzeichnet. Und in *UNIX* gibt es nicht notwendig Datei-suffixes wie das beliebte *.exe* in *Windows*. Dies sind, was die Organisation des Dateisystems angeht für uns die einzigen Unterschiede, die Sie vorerst bemerken

werden.

Es gibt eine ganze Menge von Kommandos, die Sie in der Shell angeben können. Einige kennen Sie bestimmt schon, da Sie beim Design von *command.com* übernommen wurden. In Tabelle 2 finden Sie die wichtigsten Vertreter, mit deren Hilfe Sie in der Verzeichnisstruktur Ihres Rechners navigieren können. Wichtig ist in diesem Zusammenhang, dass *UNIX* im Gegensatz zu *Windows* keine *Laufwerke* kennt. Deshalb kann *Cygwin* nichts mit der Angabe *cd a:* anfangen. Dies wird in *Cygwin* durch *cd /cygdrive/a* ersetzt. Und als letzte Bemerkung sei erwähnt, dass *UNIX*-Kommandos wie *DOS*-Kommandos Optionen und Argumente haben. Optionen werden durch die Angabe eines *-* übergeben, *ls -al* ruft das *ls*-Kommando auf mit den beiden Optionen *a* und *l*.

Kommando	Bedeutung
<i>cd</i>	„Change Directory“, wechseln in das als Argument angegebene Verzeichnis. Wird kein Argument angegeben, wechseln Sie ihn ihr Home-Verzeichnis. Beispiel: <i>cd hello</i>
<i>ls</i>	„List Directory“, auflisten des Inhalts des aktuellen Verzeichnisses. Wichtige Optionen: <i>ls -l</i> : Neben den Dateinamen weitere Details ausgeben, <i>ls -a</i> : Zeige auch versteckte Dateien an. Wird ein Verzeichnisname als Argument übergeben, wird dieses angezeigt. Beispiel: <i>ls -al</i>
<i>man</i>	„Manual“, Falls das User Manual installiert ist, kann damit eine Hilfe ausgegeben werden. Beispiel: <i>man ls</i>
<i>more</i>	Ausgabe der Datei in der Shell, mit Seitenumbruch. Beispiel: <i>more hello</i>
<i>cp</i>	„Copy“, Kopieren von Dateien oder Verzeichnissen. Wichtige Option: <i>cp -r</i> : Rekursives Kopieren von ganzen Verzeichnisbäumen. Beispiel: <i>cp quelle ziel</i>
<i>mv</i>	„Move“, Verschieben von Dateien oder Verzeichnissen. Beispiel: <i>mv quelle ziel</i>
<i>ln</i>	„Link“, Verknüpfung zwischen Dateien erstellen. Wichtige Option: <i>ln -s</i> : „soft link“, beim Löschen der Verknüpfung wird das Original nicht angetastet, im Gegensatz zu Default, dem „hard link“. Beispiel: <i>ln -s quelle ziel</i>
<i>rm</i>	„Remove“, Löschen von Dateien. Wichtige Option: <i>rm -i</i> : Interaktives Löschen, Rückfrage vor der Ausführung <i>rm -f</i> : „Force“, auch schreibgeschützte Dateien werden ohne Rückfrage gelöscht <i>rm -r</i> : Rekursives Löschen von ganzen Verzeichnisbäumen.

Tabelle 2: *UNIX*-Kommandos für die Navigation und für das Datei-Handling

Wie Sie dies vielleicht bereits von *Windows* kennen kann die Ausgabe eines Kommandos umgeleitet werden. Die Default-Kanäle für Ein- und Ausgabe heißen in *UNIX* *stdin* und *stdout* für

„Standard In“ und „Standard Out“. Geben Sie nichts an, dann ist *stdin* in der Shell die Tastatur, und *stdout* das Fenster, in dem die Shell abläuft. Umleiten geht ganz einfach mit `<` und `>`. Beispielsweise erzeugt der Befehl `ls -l > liste` eine Datei mit dem Namen *liste*, die die Ausgabe des Kommandos `ls -l` enthält. Dies ist recht nützlich, wenn Sie die Ausgaben des von Ihnen erzeugten Programms, beispielsweise zur Fehlersuche, vom Shell-Fenster in einer Datei umlenken möchten. Die Ein- und Ausgabe eines Kommandos kann in einer *Pipeline* verkettet werden. Dies gelingt durch das Zeichen `|`. Eine häufige Anwendung einer Pipeline ist der Befehl `ls -la|more`. Damit wird der Inhalt des aktuellen Verzeichnisses nicht einfach aufgelistet, sondern die Ausgabe wird an das Kommando *more* übergeben, das dafür sorgt, dass ein Seitenumbruch durchgeführt wird. Sie können danach mit Hilfe der Pausetaste durch die Liste scrollen.

Abbildung 2 zeigt den typischen Anfang einer Sitzung in *Cygwin*. Sie starten die Shell und wechseln in das Verzeichnis, in dem Sie arbeiten möchten. Sehr hilfreich ist dabei, dass Sie mit Hilfe der TAB-Taste angefangene Wörter komplettieren können. Dies hilft bei der Eingabe von langen Windows-Namen. Insbesondere dann, wenn die Namen Leerzeichen enthalten. Diese sollten Sie nach Möglichkeit bei der Namenswahl nicht verwenden!

```
brill@STING[cygdrive/h/Praktikum/Werkzeuge] >s
bash: s: command not found
brill@STING[cygdrive/h/Praktikum/Werkzeuge] >ls
Open GL VRML
brill@STING[cygdrive/h/Praktikum/Werkzeuge] >cd Open\ GL/
brill@STING[cygdrive/h/Praktikum/Werkzeuge/Open GL] >ls
beispiele
brill@STING[cygdrive/h/Praktikum/Werkzeuge/Open GL] >cd beispiele
brill@STING[cygdrive/h/Praktikum/Werkzeuge/Open GL/beispiele] >ls
first second
brill@STING[cygdrive/h/Praktikum/Werkzeuge/Open GL/beispiele] >cd second
brill@STING[cygdrive/h/Praktikum/Werkzeuge/Open GL/beispiele/second] >ls
Makefile second.cpp second.exe second.o
brill@STING[cygdrive/h/Praktikum/Werkzeuge/Open GL/beispiele/second] >ls -l
total 239
-r--r--r-- 1 brill Kein 503 Oct 8 14:58 Makefile
-r--r--r-- 1 brill Kein 4301 Oct 8 15:07 second.cpp
-rwxr-xr-x 1 brill Kein 212492 Oct 8 14:58 second.exe
-rw-r--r-- 1 brill Kein 24598 Oct 8 14:58 second.o
brill@STING[cygdrive/h/Praktikum/Werkzeuge/Open GL/beispiele/second] >make second.o
g++ -c -g second.cpp
brill@STING[cygdrive/h/Praktikum/Werkzeuge/Open GL/beispiele/second] >
```

Abbildung 2: Die Cygwin-Shell in Aktion

Tip:

Es gibt eine ganze Reihe von Einführungen in *UNIX*. Ich kann nur empfehlen, dass Sie sich bei Bedarf beim *O'Reilly*-Verlag umsehen, dort gibt es eine Unmenge von Büchern zu einzelnen *UNIX*-Kommandos. Eine gute und kurze Einführung in das Betriebssystem ist Todino, Strang, Peek: *Learning the UNIX Operating System* bei *O'Reilly* ([PTS01]). Im Rechenzentrum gibt es ein Skript zu diesem Thema vom RRZN Hannover.

3 Programmentwicklung in UNIX und CYGWIN

Diesen Abschnitt könnte man auch mit *Edit – Make – Debug* überschreiben. Dies stellt den normalen Ablauf der Programmentwicklung dar, nicht nur in *UNIX*. Sollten Sie bisher nur mit *Microsoft Visual* entwickelt haben, dann haben Sie dies zwar bereits so durchgeführt. Aber zum weiteren Verständnis der *Programmer's Workbench* hier noch mal die Stufen, die ein C oder C++-Programm durchläuft, bis ein ausführbares Programm entstanden ist:

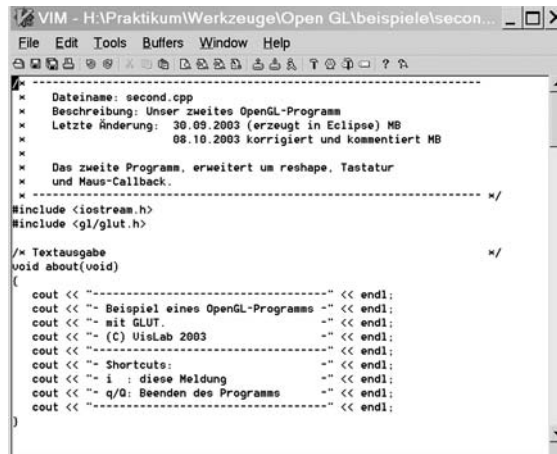
Edit: Wir erstellen die Quellcodes des Programms.

Make: Wir übersetzen, („kompilieren“) die Quellen. Das Ergebnis sind Objektdateien mit Maschinencode. Anschließend werden unsere übersetzten Quellen mit weiterem Maschinencode, der in Bibliotheken abgelegt ist zu einem ausführbaren Programm gebunden („gelinkt“).

Debug: Sollte das ausführbare Programm logische Fehler enthalten, müssen wir diese suchen. Dazu dienen *Debugger*.

3.1 Edit

Natürlich müssen wir zuerst den Quellcode erstellen oder verändern. Dazu verwenden wir einen ASCII-Editor. Davon gibt es inzwischen viele, vom *vi*, den es in jedem UNIX-System und natürlich auch unter *Cygwin* gibt über den *emacs* bis hin zu *notepad*. Arbeiten Sie mit *Cygwin*, dann können Sie in der Shell einen Editor aufrufen, oder einen beliebigen ASCII-Editor, den Sie installiert haben. In Abbildung 3 sehen Sie einen C++-Quelltext in *VIM*.



```

*   Dateiname: second.cpp
*   Beschreibung: Unser zweites OpenGL-Programm
*   Letzte Änderung: 30.09.2003 (erzeugt in Eclipse) MB
*                   08.10.2003 korrigiert und kommentiert MB
*
*   Das zweite Programm, erweitert um reshape, Tastatur
*   und Maus-Callback.
*
----- */
#include <iostream.h>
#include <gl/glut.h>

/* Textausgabe */
void about(void)
{
    cout << "-----" << endl;
    cout << "- Beispiel eines OpenGL-Programms -" << endl;
    cout << "- mit GLUT. -" << endl;
    cout << "- (C) UisLab 2003 -" << endl;
    cout << "-----" << endl;
    cout << "- Shortcuts: -" << endl;
    cout << "- i : diese Meldung -" << endl;
    cout << "- q/Q: Beenden des Programms -" << endl;
    cout << "-----" << endl;
}

```

Abbildung 3: Quelltext in *VIM*

Tip:

Verwenden Sie *notepad* nur, wenn Sie es unbedingt möchten. Insbesondere die Angewohnheit dieses Programms, die Dateien immer mit dem Suffix *.txt* versehen zu wollen ist bei der Erstellung von Quell-Code äußerst lästig. Alternativen sind die Freeware-Pakete *Proton* oder *VIM*.

Speichern Sie ein C-Programm *immer* mit dem Suffix *.c* ab, und C++-Code mit *.cpp* oder *.C*. Warum dies von Vorteil ist sehen wir weiter unten. Header-Dateien speichern Sie mit dem Suffix *.h* oder, falls Sie eine entsprechende Version der Compiler haben, ganz ohne Suffix.

3.2 Compile

Nach der Erstellung der Quelldateien folgt das Übersetzen, in der Hoffnung auf möglichst wenig Syntaxfehler. Arbeiten Sie mit *LINUX* oder *Cygwin*, dann sind die *GNU*- Compiler *gcc* und *g++* installiert. *gcc* ist der GNU C Compiler, *g++* das C++-Pendant. In jedem UNIX-System ist immer das Kommando *cc* für den C-Compiler vorhanden – UNIX ohne einen C-Compiler ist undenkbar. Im Folgenden arbeiten wir immer mit den GNU-Compilern, die Kommandos sind jedoch einfach zu verändern, sollten Sie mit *Solaris*, *AIX*, *IRIX*, *HP-UX* oder anderen UNIXen arbeiten.

Der Compiler ist für UNIX ein Kommando unter vielen und wird genauso aufgerufen wie die Kommandos in Tabelle 2: *g++ -c second.cpp* bedeutet, dass der C++-Compiler aufgerufen und die Datei *second.cpp* übersetzen werden soll. Die Option *-c* bedeutet, dass nur übersetzt und nicht versucht werden soll zu „binden“, also ein ausführbares Programm zu erzeugen. Ist das Übersetzen erfolgreich, dann gibt es im gleichen Verzeichnis, in dem auch die Datei *second.cpp* steht eine „Objektdatei“, in dem das Ergebnis des Übersetzens steht. Diese hat den gleichen Namen wie die übersetzte Quelle, allerdings den Suffix *.o*.

Jetzt haben wir also schon zwei Dateien, die Quelle und die übersetzte Quelle mit Assembler-Code. Enthält die Quelldatei ein Hauptprogramm und auch alle von Ihnen programmierten Unterfunktionen, dann können Sie die Objektdatei zu einem ausführbaren Programm „binden“. Der

Befehl, der dies ausführt und auch noch eventuell benötigte Systembibliotheken beifügt ist der „Linker“, der mit dem Befehl *ld* oder einfacher mit *g++* bzw. *gcc* aufgerufen wird. Sind Sie sicher, dass Sie beim Übersetzen keine Fehlermeldungen mehr erhalten, dann können Sie natürlich alles in einem Schritt machen und das Kommando *g++ second.cpp -o second* eingeben. Diesmal fehlt die *-c*-Option, diese wurde durch die Option *-o* ersetzt. Diese bewirkt, dass das Endergebnis, ihr ausführbares Programm, als Kommando mit dem nach *-o* angegebenen Namen, im Beispiel *second*, abgespeichert wird. Bei *LINUX* existiert dann auch eine Datei mit dem Namen *second*. *Cygwin* erstellt eine Datei *second.exe*. Haben Sie bei der Installation die DLL *cygwin1.dll* in das Windows-Verzeichnis kopiert, können Sie diese *.exe*-Datei wie gewohnt durch Doppelklick im Explorer starten. Oder Sie geben in der Shell das Kommando *second* ein, auch dies führt Ihr Programm aus. Geben Sie die Option *-o* nicht an, dann wird natürlich trotzdem ein ausführbares Kommando erzeugt. Allerdings erhalten Sie in *Cygwin* den seltsamen Namen *a.exe* und bei anderen *UNIX*-Systemen die Datei *a.out*. Also ist die Option *-o* immer zu empfehlen.

Sollten Sie noch Bibliotheken oder andere Objektdateien benötigen, um ein ausführbares Programm zu binden können sie dies mit der Option *-l* angeben. Die Option *-lm* beim Aufruf von *g++* führt dazu, dass die Bibliothek *libm.a* mit den mathematischen Funktionen aus der Header *math.h* angebunden wird. Objektdateien können Sie einfach in der richtigen Reihenfolge dazugeben. Das Kommando *g++ second.o -o second helper.o -lm* geht davon aus, dass ihr Hauptprogramm in der Datei *second.cpp* bereits erfolgreich übersetzt wurde. Das ausführbare Kommando soll den Namen *second* erhalten, und für das Binden werden noch Funktionen aus der überfalls bereits übersetzten Datei *helper.o* und der mathematischen Bibliothek */usr/lib/libm.a* benötigt.

Tip:

In *UNIX* gibt es neben den statischen Bibliotheken, die den Suffix *.a* tragen natürlich auch die Möglichkeit, dynamisch zu binden. Diese Bibliotheken, die Ihnen aus *Windows* durch den Suffix *.dll* bekannt sind, heißen *shared objects* und haben den Suffix *.so*.

Der Linker sucht die Bibliothek *libm.a* in einigen Defaultverzeichnissen wie */lib*, */usr/lib* oder im aktuellen Verzeichnis „.“, in dem gerade gearbeitet wird. Mit der Option *-L* können Sie weitere Verzeichnisse angeben, in der nach Bibliotheken gesucht werden soll. Ähnlich gibt es Defaultverzeichnisse, in denen der C-Präprozessor nach Header-Dateien sucht, die durch *#include* angegeben werden. In Normalfall ist dies mindestens das Verzeichnis */usr/include* und das aktuelle Verzeichnis. Durch die Option *-I* können Sie weitere Verzeichnisse angeben, in denen Header-Dateien stehen. Tabelle 3 fasst diese Optionen nochmals zusammen.

Tip:

Die angegebenen Compiler-Optionen gelten nicht nur für *C*- oder *C++*-Compiler, sondern als Quasi-Standard für *alle* Compiler auf einem *UNIX*-System. Ein Pascal-Compiler wird typischerweise mit *pc* oder *pcc*, ein FORTRAN-Compiler mit *f77* oder *f90*, der *Java*-Compiler mit *javac*, ein *Ada*-Compiler mit ...

3.3 Make

Das klingt ja reichlich kompliziert, wie in *Cygwin* oder allgemein in *UNIX* die Compiler zu bedienen sind. Haben Sie eine kommerzielle *UNIX*-Variante vor sich, dann ist die Chance groß, dass dazu auch eine kommerzielle und professionelle Entwicklungsumgebung dazugehört. Mit *Eclipse* und dem *CDT* steht eine integrierte Umgebung für die *Java*- oder *C++*-Entwicklung ebenfalls zur Verfügung, unter *Windows* und unter *LINUX*.

Aber es gibt noch eine weitere Variante, das *make*-Kommando. Das Betriebssystem *UNIX* ist von Software-Entwicklern gestaltet worden. Die Herren *Kernighan* und *Ritchie* haben nicht nur ihre

Option	Bedeutung
-c	„Compile“, Nur übersetzen und erzeugen einer Objekt-Datei. Beispiel: <code>g++ -c hello.cpp</code>
-g	„Debug Information“, Erzeugen von Debug-Informationen im Ergebnis. Beispiel: <code>g++ -g hello.cpp</code>
-o	„Output“, Die Ausgabe der Compilers wird unter dem angegebenen Namen abgespeichert. Beispiel: <code>g++ hello.cpp -o hello</code>
-I	Weitere Verzeichnisse, in denen der Präprozessor nach Header-Dateien suchen soll. Beispiel: <code>g++ -c hello.cpp -I. -I/usr/local/include -I../include</code>
-L	Weitere Verzeichnisse, in denen der Linker nach Bibliotheken suchen soll. Beispiel: <code>g++ -c hello.cpp -L. -L/usr/local/lib -L../lib</code>
-l	Angabe von Bibliotheken, die angebunden werden sollen. Beispiel: <code>g++ -c hello.cpp -L/usr/local/lib -lglut32 -lglu32 -lopengl32</code>

Tabelle 3: Die wichtigsten Optionen der UNIX-Compiler

Spuren in der Programmiersprache C, sondern auch in UNIX hinterlassen. Mit *make* wird UNIX wirklich zu einer Programmer's Workbench!

Tip:

Makefiles und ein *make*-Kommando stehen Ihnen in einer *Eingabeaufforderung* auch in Windows zur Verfügung, wenn Sie *Microsoft Visual* oder vergleichbare Produkte installiert haben. Suchen Sie nach der Datei *nmake.exe*; der Name steht für „new make“.

Rekapitulieren wir nochmals, welche Schritte ein Quellcode durchläuft, bis wir das Programm ausführen können. Angenommen, der Quellcode liegt in den beiden Dateien *second.cpp* und *helper.cpp* und wir benötigen noch die mathematische Bibliothek *libm.a*, dann müssen Sie, angenommen es gibt keine Syntaxfehler, die folgenden Kommandos eingeben:

```
g++ -c helper.cpp
g++ -c second.cpp
g++ -o second second.o helper.o -lm
```

Das geht viel kürzer – und es wird jetzt auch klar, warum die Suffixes für Quellen und Objekte festgelegt waren. Das *make*-Kommando „weiß“, dass Sie bei der Eingabe von

```
make hello
```

offensichtlich eine Datei *hello.cpp* haben und daraus das ausführbare Kommando *hello* machen wollen. Probieren sie es aus, *make* führt die nötigen Aufrufe des Compilers eigenständig aus. Natürlich weiß *make* nichts von den Abhängigkeiten wie in unserem Beispiel, in dem die Quelle in den beiden Dateien *second.cpp* und *helper.cpp* verteilt waren. Dieser Umstand kann aber durch einen *Makefile* geholfen werden. *make* sucht bei einem Aufruf wie `make hello` im aktuellen Verzeichnis nach einer Datei *Makefile*, bei Misserfolg nach *makefile*. Werden beide nicht gefunden, und wurde auch durch eine Option keine andere Eingabe-Datei angegeben, dann wird mit Hilfe von impliziten Regeln versucht, eine ausführbare Datei zu erzeugen. In unserem Beispiel mit `make hello` kommen zwei implizite Regeln zum Tragen: einmal, dass erst mal aus einer *.cpp* Datei mit Hilfe des C++-Compilers eine *.o* Objektdatei gemacht werden soll; und anschließend diese Objektdatei zu einem ausführbaren Kommando gebunden wird.

Wie könnte nun ein Makefile für unser Beispiel der beiden Quellen *second.cpp* und *helper.cpp* aussehen? Das ausführbare Programm *second* hängt von den beiden Objektdateien *second.o* und

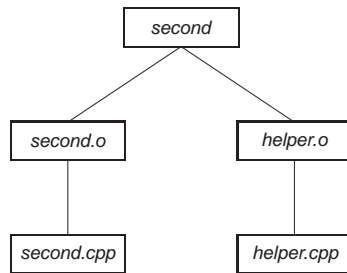


Abbildung 4: Die Abhängigkeiten für unser ausführbares Programm `second`

von `helper.o` ab; diese beiden wiederum von ihren Quelldateien. Das kann man sich als Baum vorstellen wie in Abbildung 4. `make` kennt sogenannte Abhängigkeitsregeln oder *Dependencies*, mit deren Hilfe man angeben kann, wie die einzelnen Bausteine zusammenhängen, also den Baum aus Abbildung 4 definieren kann:

```

# Das ausführbare Kommando second hängt von second.o und
# helper.o ab:
second : second.o helper.o
        g++ -o second second.o helper.o -lm

# second.o hängt von second.cpp ab:
second.o : second.cpp
          g++ -c second.cpp

# helper.o hängt von helper.cpp ab:
helper.o : helper.cpp
          g++ -c helper.cpp
  
```

Eine Dependency ist eine Zeile, bei der auf der linken Seite eines Doppelpunktes das *Target*, das *Zielobjekt* der Abhängigkeitsregel steht. Im Beispiel oben also `second` oder `helper.o`. Rechts vom Doppelpunkt stehen die Objekte, von denen das Target abhängt. `make` überprüft das Dateidatum von Target und Eingabeobjekten. Ist mindestens eines der Eingabeobjekte neuer, dann muss offensichtlich das Zielobjekt neu „gemacht“ werden – daher der Name `make` für dieses Kommando. In der auf die Dependency folgenden Kommando-Zeilen werden die Shell-Kommandos angegeben, die die Eingabeobjekte zum Target machen. Dies können mehrere Zeilen sein. Wichtig ist, dass jede dieser Kommandos mit einem Tabulator beginnt. `make` erkennt, dass alle nötigen Kommandos abgearbeitet wurden durch die erste Zeile, die nicht mit einem Tabulator erkennt.

Tip:

Ein beliebter Fehler ist, statt des Tabulator in den Kommandozeilen Leerzeichen einzufügen. Sollte ein Makefile nicht funktionieren, überprüfen Sie die Syntax der Kommandozeilen!

Beim Aufruf `make` wird das erste in der Datei angegebene Target gesucht und dann überprüft, was genau durchgeführt werden muss. Sind beispielsweise beide Objektdateien „up to date“ (sie müssen also eine Dateidatum neuer als die Quellen haben), dann wird nur gebunden. Hat beispielsweise `helper.cpp` ein neueres Datum als `helper.o`, dann muss offensichtlich das Kommando aufgerufen werden, das `helper.o` neu erzeugt. Nun wirkt die Regel für das Ziel `second`, auch dieses Target wird neu erzeugt. Sie sehen, `make` ist in der Lage, rekursiv eine große Menge von Abhängigkeiten zu überprüfen und nur die Aktionen durchzuführen, die notwendig sind.

Sie können `make` auch dazu verwenden, dezidiert ein Ziel zu überprüfen. Sieht das *Makefile* so aus wie in unserem Beispiel für das Ziel `second`, dann überprüft der Aufruf `make second.o`, ob

die Objektdatei neu erstellt werden muss. Ist dies der Fall, werden nur die Abhängigkeitsregeln überprüft, die hierarchisch unterhalb der Regel für *second.o* liegen.

Tip:

In den Kommandozeilen eines Makefiles sind *alle* ausführbaren Kommandos des Betriebssystems zulässig. Das wird häufig dazu verwendet, um Software zu installieren oder Dateien zu kopieren.

In einem *Makefile* können auch Targets vorkommen, die keine Abhängigkeiten besitzen. Ein typisches Beispiel sind Ziele mit dem Namen *clean* oder *all*. *all* wird häufig als erstes Ziel in einem Makefile angegeben; und mit *clean* ist ein „Aufräumen“ verbunden, also ein Löschen von Zwischendateien wie den Objektdateien:

```
# Zwei ausführbare Programme in einem Makefile ...
all: first second

first : .....

second : ....

# Aufräumen
clean :
    /bin/rm -f *.o
```

Tip:

Bei der Erstellung eines neuen Projekts in *Eclipse* benötigen Sie neben den Quellen auch einen Makefile. Dabei geht *Eclipse* davon aus, dass mindestens die beiden Targets *all* und *clean* enthalten sind. Geben Sie in *Eclipse* das Kommando *Rebuild All* an, dann wird intern ein *make all* durchgeführt.

make kennt darüberhinaus sogenannte *Makros*. Damit können Sie in einen Makefile Variablen einführen und beispielsweise Pfade oder Bibliotheksnamen zentral definieren. *Makros* bestehen aus eine Zeile mit einem Gleichheitszeichen:

```
# ABC definieren
ABC = XYZ
# Jetzt können wir ${ABC} verwenden
FILE = Text.${ABC}
```

Nach der Definition wird das Makro *ABC* durch *\${ABC}* verwendet.

Tip:

make gibt keine Fehlermeldung aus, falls ein Makro verwendet wird bevor es definiert ist. In diesem Fall kommt der Null-String zur Anwendung. Eine beliebte Fehlerquelle ...

Neben der Definition von Makros im Makefile ist es auch möglich, globale Umgebungsvariablen zu definieren. Auf diese Weise kann mit der gleichen Syntax wie auf Makros zugegriffen werden. Ein Makro mit dem gleichen Namen überschreibt eine Umgebungsvariable! *make* kennt darüberhinaus eine Vielzahl von vordefinierten Makros, die natürlich im Makefile überschrieben werden können. Beispielsweise gibt es häufig das Makro *\$(CXX)*, das mit dem Aufruf des C++-Compilers belegt ist. Wollen wir sicher gehen, dass der GNU-Compiler verwendet wird, kann dies durch die Zeile

```
# C++-Compiler festlegen
CXX = g++
```

durchgeführt werden. Diese Definitionen sind jedoch systemabhängig; mit einem Aufruf des Manuals (*man make*) erhalten Sie eine entsprechende Dokumentation.

Um Kommandozeilen möglichst generisch schreiben zu können sind einige interne Makros sehr hilfreich. Das Makro `@` ist in einem Makefile immer implizit definiert und wird bei der Ausführung einer Kommandozeile mit dem Target besetzt. Damit kann unser Beispiel für das Binden des ausführbaren Programms *second* wie folgt geschrieben werden:

```
# C++-Compiler festlegen
CXX = g++
# Debuggen? Falls nein, einfach diese Definition verändern!
DEBUG = -g
# Optimieren? Fall ja, einfach -O in dieses Makro!
OPTIMIZE =
# Alle Compiler-Optionen zusammenfassen und @ verwenden
CXXFLAGS = ${OPTIMIZE} ${DEBUG} -o ${@}

second : second.o
    ${CXX} ${CXXFLAGS} second.o helper.o -lm
```

Das Makro `?` wird in einer Kommandozeile durch eine Liste *der* Eingabeobjekte ersetzt, die neuer als das Target sind und die Kommandozeile aktiviert haben. Dadurch können wir die letzte Zeile in unserem Beispiel weiter verallgemeinern zu

```
second : second.o
    ${CXX} ${CXXFLAGS} $? helper.o -lm
```

Analog können wir selbstverständlich ein Makro `{LDFLAGS}` definieren, das für das Binden die entsprechenden Compiler-Optionen enthält. Und alle Bibliotheken und Objektdateien, die wir benötigen werden häufig ebenfalls zu einem Makro zusammengefasst. Die einzelnen Regeln sehen dann etwas gewöhnungsbedürftig aus, aber Sie können die einzelnen Bausteine zentral zu Beginn des Makefiles ändern, und neue Regeln sehr einfach mit „Copy and Paste“ einfügen. Der folgende Ausschnitt zeigt den Aufbau eines Makefiles, wie Sie ihn in den Übungen zu Visualisierungen oder Computer-Animation und -Visualisierung erhalten werden:

```
# Makros für Compiler, Compiler-Optionen
CXX          = g++
DEBUG        = -g
OPTIMIZE     =
CXXFLAGS    = -I. -I/usr/local/include ${DEBUG} ${OPTIMIZE}
LDFLAGS     = ${CXXFLAGS} -o ${@} -L/usr/local/lib

# Die OpenGL-Bibliotheken:
OGLLIBS     = -lglut32 -lglu32 -lopengl32

all : first second

first : first.o
    ${CXX} -o ${@} ${CXXFLAGS} $? ${OGLLIBS}

first.o : first.cpp
    ${CXX} -c ${CXXFLAGS} $?

second : second.o
    ${CXX} -o ${@} ${CXXFLAGS} $? ${OGLLIBS}

second.o : second.cpp
```

```

    ${CXX} -c ${CXXFLAGS} $?

.DEFAULT :
    echo "Keine Regel gefunden!"

clean:
    touch _._o _~
    /bin/rm -f *.o

```

Tip:

Eine hervorragende Einführung ist das Buch Oram, Talbot: *Managing Projects with make* aus dem O'Reilly Verlag ([TO91]).

3.4 Debug

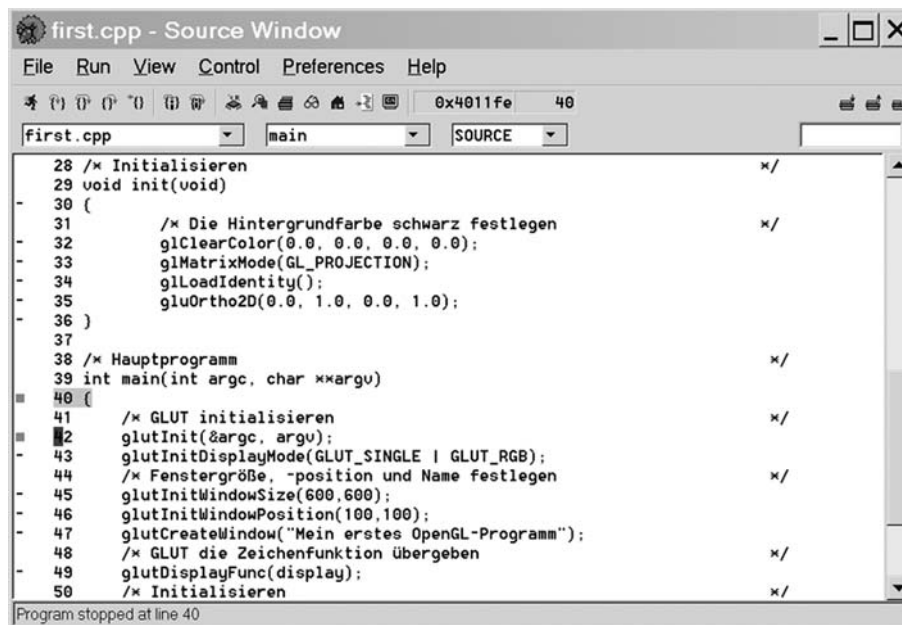
Der normale Lauf der Dinge ist, dass Sie je nach Übung irgendwann einmal ein ausführbares Programm erhalten haben. Die Anzahl der Syntax-Fehler geht bei genügend Übung gegen Null. Es bleiben die logischen Fehler, die Software tut noch nicht das, was sie eigentlich soll, oder stürzt unmotiviert ab – in diesem Fall wächst sicher Ihr Verständnis für die Entwickler bei Microsoft ...

Dann bleibt nur übrig, sich auf Fehlersuche zu begeben. Dies ist natürlich durch den Einbau beliebig vieler Ausgabe-Statements in Form von `printf` oder `cout` möglich. Dies vermeiden Sie möglichst, Sie vergessen garantiert einige nach der erfolgreichen Fehlersuche, und Ihre Benutzer wundern sich über unmotivierte Ausgaben. Besser ist die Verwendung eines Debuggers. Der Name dieser Spezies kommt übrigens von der Fehlersuche der Entwickler der ersten Computer. Die Anekdote besagt, dass ein Programm immer abstürzte und die Entwickler in der Logik der Software partout keinen Fehler feststellen konnten. Der Grund wurde schließlich trotzdem gefunden in Form einer Motte, die sich auf eine der Röhren niedergelassen und dadurch einen Kurzschluss hervorgerufen hatte. Seitdem heißt (angeblich) die Fehlersuche in einem Programm Debuggen von „to debug“, was übersetzt so viel wie „entkäfern“ heißt.

Aktion	Bedeutung
R	„run“, das Programm starten.
C	„continue“, das Programm bis zum nächsten Breakpoint weiterlaufen lassen.
F	„finish“, das Programm bis zum Ende weiterlaufen lassen.
N	„next“, zur nächsten ausführbaren Anweisung im Quellcode gehen. Funktionsaufrufe sind eine Anweisung.
S	„step“, zur nächsten ausführbaren Anweisung im Quellcode gehen. Springt in eine Funktion hinein.
Zeilennummer anklicken	Diese Zeile soll ein Breakpoint sein.

Tabelle 4: Die wichtigsten Kommandos zur Bedienung des *gdb*

Haben Sie die GNU-Compiler installiert, dann ist mit ziemlicher Sicherheit auch der GNU-Debugger *gdb* installiert. Je nach Installation erhalten wir einen interaktiven Debugger durch das Kommando *gdb* oder *insight*. Vorausgesetzt, das Programm *second* wurde mit *-g*-Option übersetzt, dann ergibt der Aufruf *insight second* ein Fenster wie in Abbildung 5. Darin können sie zeilenweise durch das Programm gehen, sich die Variablenwerte anzeigen lassen, Variablenwerte verändern, Breakpoints (Stellen, an denen die Programmausführung angehalten werden soll) setzen und vieles mehr. Die Bedienung ist ähnlich wie in Visual C++. In Tabelle 4 finden Sie die grundlegenden Shortcuts und Funktionen.



```
first.cpp - Source Window
File Run View Control Preferences Help
0x4011fe 40
first.cpp main SOURCE
28 /* Initialisieren */
29 void init(void)
30 {
31     /* Die Hintergrundfarbe schwarz festlegen */
32     glClearColor(0.0, 0.0, 0.0, 0.0);
33     glMatrixMode(GL_PROJECTION);
34     glLoadIdentity();
35     gluOrtho2D(0.0, 1.0, 0.0, 1.0);
36 }
37
38 /* Hauptprogramm */
39 int main(int argc, char **argv)
40 {
41     /* GLUT initialisieren */
42     glutInit(&argc, argv);
43     glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
44     /* Fenstergröße, -position und Name festlegen */
45     glutInitWindowSize(600, 600);
46     glutInitWindowPosition(100, 100);
47     glutCreateWindow("Mein erstes OpenGL-Programm");
48     /* GLUT die Zeichenfunktion übergeben */
49     glutDisplayFunc(display);
50     /* Initialisieren */

```

Program stopped at line 40

Abbildung 5: Der *insight*-Debugger in *Cygwin*

4 Ausblick

Die Arbeit mit einer Shell und Makefiles erscheint auf den ersten Blick umständlich, verglichen mit der Mächtigkeit und dem Komfort einer interaktiven Entwicklungsplattform wie *Microsoft Visual* oder *Eclipse*. Allerdings können sie davon ausgehen, dass im Hintergrund exakt die Mechanismen greifen, die in diesen Text beschrieben sind. Schon dies stellt eine Motivation dar, hinter die Kulissen zu schauen. Falls Sie nur Konsolenanwendungen ohne ein GUI implementieren, ist nach etwas Übung die Arbeit mit einer Shell und einem Makefile deutlich effizienter als mit einem „Klicker-Werkzeug“.

Viel Spass bei der Software-Entwicklung unter *LINUX* oder *Cygwin* – Welcome to the real world!

Literatur

[PTS01] PEEK, JERRY, TODINO, GRACE und STRANG, JOHN: *Learning the UNIX Operating System*. O'Reilly, 2001.

[TO91] TALBOT, STEVE und ORAM, ANDREW: *Managing Projects with Make*. O'Reilly, 1991.