23

Effektive Programmerzeugung

Dieses Kapitel behandelt die folgenden Themen:

- Automatische Ermittlung von Abhängigkeiten
- Makefile für Verzeichnisbäume
- Automatische Erzeugung von Makefiles
- Erzeugen von Bibliotheken
- Code Bloat bei der Instanziierung von Templates vermeiden
- GNU Autotools
- Alternative CMake

Dieses Kapitel beschreibt ergänzende und mächtigere Techniken für *make* als in der Einführung vorgestellt. Wie dort wird auch für die Beispiele dieses Kapitels die Klasse `Rational` von Seite 164 ff. benutzt. Es gibt die Dateien *main.cpp*, *rational.h* und *rational.cpp*.

Die Grundlagen zu Make finden Sie in Kapitel 17.

23.1 Automatische Ermittlung von Abhängigkeiten

Von welchen Header-Dateien eine cpp-Datei abhängig ist, lässt sich durch einen Blick auf die `#include`-Anweisungen der cpp-Datei feststellen. Die `#include`-Anweisungen werden ohnehin bei der Compilation durch den C++-Präprozessor gelesen – der hat die benötigten Informationen! Die Entwickler des GNU-C++-Compilers haben daran gedacht, diese Informationen zur Verfügung zu stellen. Dazu wird der Compiler mit der Option -M aufgerufen, die alle Abhängigkeiten in einer Form ausgibt, die für ein Makefile geeignet ist. Um die System-Header-Dateien auszublenden, bietet sich die Option -MM an. So ergibt der Aufruf `g++ -MM rational.cpp > rational.d` die Datei *rational.d* mit dem Inhalt

```
rational.o: rational.cpp rational.h
```

Die Optionen -M und -MM implizieren die Option -E, die dafür sorgt, dass der Compiler nach Aufruf des Präprozessors nichts mehr tut, also weder compilieren noch linken. Es ist üblich, die Abhängigkeiten in Dateien mit der Endung .d (für dependency) zu speichern, hier also *rational.d*, und diese Dateien mit `include` im Makefile einzuschließen. Das Minuszeichen vor `include` unterdrückt Meldungen bei fehlenden d-Dateien – ein normaler Zustand, schließlich sollen sie ja bei Bedarf erzeugt werden. In der oben gezeigten Datei *rational.d* wird berücksichtigt, dass eine Änderung in der Datei *rational.h* zur Neubildung von *rational.o* führen muss, nicht aber, dass auch *rational.d* vielleicht neu berechnet werden müsste. Es könnte ja sein, dass zum Beispiel eine `#include"neu.h"`-Anweisung in *rational.h* aufgenommen wurde. Eine Änderung von *neu.h* müsste dann auch zur Neucompilation führen. Aus diesem Grund sollte *rational.d* selbst auch von *rational.h* abhängen, das heißt, *rational.d* sollte besser so aussehen:

```
rational.d rational.o: rational.cpp rational.h
```

Man könnte nun die Zeichenkette »rational.d«, gefolgt von einem Leerzeichen, in die gleichnamige Datei schreiben, etwa

```
%.d: %.cpp
→@echo -n $@ " " > $@
→$(CXX) -MM >> $@ $<
```

und die Ausgabe von `$(CXX) -MM` anhängen. Das Symbol → steht für das Tabulatorzeichen. Zur Erinnerung: `$@` gibt das Ziel und `$<` die erste Abhängigkeit an. -n unterdrückt den Zeilensprung. Die zwei Aufrufe der Shell könnten zusammengefasst werden:

```
→@echo -n $@ " " > $@ && $(CXX) -MM >> $@ $<
```

`&&` ist die Verkettung von Befehlen in der Shell. Es gibt aber auch die Option -MT, mit der der Teil vor dem Doppelpunkt definiert werden kann:

```
→$(CXX) -MM -MT "$@ $(patsubst %.d,%.o,$@)" -MF $@ $<
```

-MT schreibt in unserem Beispiel erst »rational.d«, dann »rational.o« als Ergebnis der Ersetzung mit `patsubst`. Die Anführungszeichen sorgen dafür, dass alles dazwischen zur

Option -MT gehört. -MF gibt den Namen der zu erzeugenden Datei an, der identisch mit dem Ziel ist. Am Ende der Aktion folgt der Name der zu analysierenden Datei `$<`, also der entsprechende cpp-Dateiname. Vermutlich ist diese Lösung effizienter als die vorherigen, weil die d-Datei nur einmal angefasst wird. Manche wählen noch standardmäßig die Option -MG in der Aktion für d-Dateien. Sie bewirkt, dass fehlende Header-Dateien nicht als Fehler gelten. Dies ist dann wichtig, wenn Header-Dateien noch in einem anderen Schritt generiert werden sollen, etwa bei der Verarbeitung von Grammatiken mit den Programmen *yacc* oder *bison*. Ich empfehle, die Option -MG nicht einzusetzen, um Fehler schneller erkennen zu können – es sei denn, man braucht sie aus den genannten Gründen wirklich. Die Datei *m4.mak* auf der Basis von *m3.mak* von Seite 522 zeigt, wie es geht:

Listing 23.1: Makefile mit automatischer Generierung der Abhängigkeiten

```
# cppbuch/k23/abhaengigkeiten/m4.mak
.PHONY: all clean
CXX := g++
CXXFLAGS := -c -g -Wall
LDFLAGS := -g
objs := $(patsubst %.cpp,%.o,$(wildcard *.cpp))
deps := $(objs:.o=.d)
all: projekt.exe
-include $(deps)

projekt.exe: $(objs)
	$(CXX) $(LDFLAGS) -o $@ $^

%.d: %.cpp
	$(CXX) -MM -MT "$@ $(patsubst %.d,%.o,$@)" -MF $@ $<

%.o: %.cpp
	$(CXX) $(CXXFLAGS) $<

clean:
	rm -f $(objs) $(deps)
```

Die Definition der Variablen `deps` zeigt eine abkürzende Art der Musterersetzung, die Langform ist `deps := $(patsubst %.o,%.d,$(objs))`. Eine Besonderheit gibt es beim Löschen der Dateien zu beobachten: Wenn `make -f m4.mak clean` *zweimal* direkt nacheinander aufgerufen wird, werden die d-Dateien vor dem zweiten Aufruf erst neu erzeugt, dann gelöscht. Der Grund liegt in der `include`-Anweisung, die unabhängig von jedem Ziel ausgewertet wird, d.h. auch bei `clean`.

23.1.1 Getrennte Verzeichnisse: src, obj, bin

Bei größeren Projekten ist es üblich, Quellen- und Objektverzeichnisse zu trennen. Ein Vorteil ist, dass das Quellenverzeichnis nicht durch generierte Dateien »verunreinigt« wird und ohne Platzvergeudung oder Löschoperationen gesichert werden kann. Es wird oft *src* (für das engl. *source*) genannt. Die Verzeichnisnamen sind natürlich frei wählbar, aber es gibt Konventionen. Daran orientiert, nenne ich das Verzeichnis, in dem die Objektdateien liegen, *obj* und das Verzeichnis mit der ausführbaren Datei *bin*. Die Abhängigkeitsdateien (Endung .d) sind im Verzeichnis *obj*, weil auch sie generierte Dateien sind und jederzeit

neu erzeugt werden können. Damit ergibt sich nach Ablauf des *make*-Kommandos das folgende, mithilfe des Unix-Programms `tree` erzeugte Abbild der Verzeichnisstruktur:

```
.                              Projektebene
|-- bin
|   '-- projekt.exe            Executable
|-- makefile
|-- obj
|   |-- main.d
|   |-- main.o
|   |-- rational.d
|   '-- rational.o
'-- src
    |-- main.cpp
    |-- rational.cpp
    '-- rational.h
```

Die Verzeichnisse *obj* und *bin* werden bei Bedarf neu angelegt. Das folgende *makefile* leistet das Gewünschte, ohne dass die cpp-Dateien und ihre Abhängigkeiten spezifiziert werden müssten. Es müssen nur die Verzeichnisnamen definiert werden:

Listing 23.2: Makefile mit Generierung der Abhängigkeiten und getrennten Verzeichnissen

```
# cppbuch/k23/verzTrennung/makefile
CXX := g++
CXXFLAGS := -c -g -Wall
LDFLAGS := -g
# Festlegen der Verzeichnisnamen
SRCDIR := src
OBJDIR := obj
BINDIR := bin

EXEFILE := projekt.exe

cppfiles := $(wildcard $(SRCDIR)/*.cpp)
objects := $(subst $(SRCDIR)/, $(OBJDIR)/, $(cppfiles:.cpp=.o))
deps := $(objects:.o=.d)

.PHONY: all clean

all: $(BINDIR)/$(EXEFILE)
-include $(deps)

$(OBJDIR)/%.d: $(SRCDIR)/%.cpp
	mkdir -p $(@D)
	$(CXX) -MM -MT "$@ $(patsubst %.d,%.o,$@)" -MF $@ $<

$(OBJDIR)/%.o: $(SRCDIR)/%.cpp
	@echo compiling $< ...
	$(CXX) $(CXXFLAGS) $(INCLUDE) $< -o $@

$(BINDIR)/$(EXEFILE): $(objects)
	mkdir -p $(BINDIR)
	$(CXX) -o $@ $^ $(LDFLAGS)
```

```
clean:
	$(RM) -r -f $(OBJDIR)
	$(RM) -r -f $(BINDIR)
```

Was geschieht im Einzelnen? Zunächst einmal werden die cpp-Dateien mit dem von oben bekannten `wildcard`-Kommando ermittelt. Das Quellenverzeichnis `$(SRCDIR)/` muss angegeben werden.

Im nächsten Schritt wird zur Ermittlung der Objektdateien `subst` eingesetzt. `subst` (für Substitution oder Ersatz) hat drei Argumente: zu ersetzender String, Ersatzstring, zu modifizierender Text. Der Text besteht aus der Liste der Namen der cpp-Dateien, deren Endung .cpp durch .o ersetzt wird. In dieser Liste wird die Zeichenkette src/ durch obj/ ersetzt. Das Ergebnis ist die Liste der zu erzeugenden Objektdateien. Daraus wird in der nächsten Zeile die Liste der .d-Dateien erzeugt, indem die Endung .o durch .d ersetzt wird.

`mkdir -p` erzeugt das Verzeichnis, falls es noch nicht vorhanden ist. Dabei ist `@D` der Verzeichnisanteil (D = directory) des Ziels. Das Ziel `clean` löscht die Verzeichnisse mit den generierten Dateien mit dem Programm, das in der Variablen `$(RM)` definiert wurde. Die Voreinstellung ist `rm`, der Unix-Befehl zum Löschen. Der Rest ist wie auf Seite 603.

23.2 Makefile für Verzeichnisbäume

Ergänzend zur oben beschriebenen Trennung von Verzeichnissen ist es üblich, dass es Unterverzeichnisse entsprechend den einzelnen Modulen gibt. Jedem Modul kann ein Namespace zugeordnet werden, sodass die Verzeichnis-Hierarchie die Namespace-Hierarchie abbilden kann. Das Verzeichnis der Objektdateien spiegelt die Struktur des Quellenverzeichnisses wieder. Bezogen auf das bisher betrachtete einfache Beispiel, aufgeteilt in zwei Module `appl` und `rational`, könnte der Verzeichnisbaum wie folgt aussehen (ohne Makefiles):

```
Projekt
|-- bin                  // enthält Executable
|   '-- projekt.exe
|-- dist
|   '-- gezippt.zip      // Distribution
|-- obj                  // automatisch generiertes Verzeichnis
|   |-- appl
|   |   |-- main.d
|   |   '-- main.o
|   '-- rational
|       |-- rational.d
|       '-- rational.o
'-- src                  // Quellverzeichnis
    |-- appl             // enthält Modul appl, die Anwendung
    |   '-- main.cpp
    '-- rational         // enthält Modul rational
```

```
|-- rational.cpp
'-- rational.h
```

Den Zusammenhang mit Namespaces zeigen die folgenden Code-Fragmente, wobei auch die Schachtelung von Namespaces gezeigt wird (obwohl es hier nur das zu schachtelnde Modul `rational` gibt):

```
// cppbuch/k23/dirTree/src/rational/rational.h
#ifndef PROJEKT_RATIONAL_H
#define PROJEKT_RATIONAL_H
namespace projekt {
namespace rational {
  class Rational {
     public: // Rest weggelassen
  };  // Klasse Rational
}} // Namespaces
#endif
```

```
// cppbuch/k23/dirTree/src/rational/rational.cpp
#include"rational.h"
namespace projekt {
namespace rational {
  // hier folgen alle Funktionsdefinitionen
}} // Namespaces
```

```
// cppbuch/k23/dirTree/src/appl/main.cpp
#include"../rational/rational.h"
using projekt::rational::Rational;
int main() {
    Rational a,b; // usw. (Rest von main() weggelassen)
}
```

Oft enthalten Verzeichnisse unterhalb der obersten Ebene des Quellenverzeichnisses selbst wieder Makefiles, die von der jeweils oberen Ebene aufgerufen werden, rekursiver Aufruf genannt. Diese Makefiles müssen natürlich angelegt werden. Wenn viele ausführbare Programme zu erzeugen sind, ist das ein sinnvoller Weg. Wenn es aber wie hier um nur ein ausführbares Programm *projekt.exe* geht, genügt ein einziges Makefile auf Projektebene. Makefiles in Unterverzeichnissen werden nicht gebraucht. Beide Möglichkeiten werden in den folgenden Abschnitten demonstriert.

23.2.1 Rekursive Make-Aufrufe

Der Einfachheit halber wird hier auf die automatische Bestimmung der Abhängigkeiten und die Spiegelung der Struktur des Quellenverzeichnisses im Verzeichnisbaum der Objektdateien verzichtet. Wie in [Mill] gezeigt, können rekursive Make-Aufrufe Probleme bei gegenseitigen Abhängigkeiten verursachen. Weil es hier nur um die Darstellung der Wirkungsweise rekursiver Make-Aufrufe geht, sie aber nicht immer empfehlenswert sind, erscheint die Vereinfachung gerechtfertigt. Im einfachen Fall könnte das Makefile auf Projektebene wie folgt aussehen:

```
export CXX := g++
export CXXFLAGS := -g -Wall -c
```

```
export LDFLAGS := -g
all:
⟶|$(MAKE) -C src/rational
⟶|$(MAKE) -C src/appl
clean:
⟶|$(MAKE) -C src/appl clean
```

Die `export`-Anweisung sorgt dafür, dass die Variablen in den Unterverzeichnissen bekannt sind. Die Aktionen sind nichts als der Aufruf von *make* in den jeweiligen Unterverzeichnissen. `$(MAKE)` ist das auf Projektebene aufgerufene Make-Programm – diese Art des Aufrufs garantiert, dass kein anderes Make-Programm (sofern vorhanden) in den Unterverzeichnissen aufgerufen wird. Alternativ wäre auch das Kommando `cd src/rational && $(MAKE)` usw. möglich gewesen. Weil jede ⟶|-Zeile an eine eigene Shell-Instanz weitergegeben wird, wirkt sich der Wechsel in ein anderes Verzeichnis nicht auf andere Aktionen aus. Das Ziel `clean` wird nur an das Verzeichnis weitergereicht, in dem gelinkt wird (siehe unten), weil dazu alle benötigten Objektdateien referenziert werden müssen. Diese Information kann für das Löschen verwendet werden. Es folgen die Makefiles für die Unterverzeichnisse:

Listing 23.3: Makefile für Unterverzeichnis rational

```
# cppbuch/k23/dirTreeRec/src/rational/makefile
objs := rational.o
all: $(objs)
%.o : %.cpp
⟶|@echo compiling $< .......
⟶|-$(CXX) $(CXXFLAGS) $< -o $@
```

Listing 23.4: Makefile für Unterverzeichnis appl

```
# cppbuch/k23/dirTreeRec/src/appl/makefile
objs := main.o ../rational/rational.o
exe := ../../bin/projekt.exe
all: $(exe)
%.o : %.cpp
⟶|@echo compiling $< .......
⟶|-$(CXX) $(CXXFLAGS) $< -o $@
$(exe): $(objs)
⟶|mkdir -p $(@D)
⟶|@echo linking $^
⟶|$(CXX) -o $@ $(objs) $(LDFLAGS)
clean:
⟶|$(RM) $(objs)
```

In der vorletzten Regel wird `$(@D)` verwendet, der Verzeichnisanteil (D = directory) des Ziels. Dementsprechend bezeichnet die automatische Variable `@F` den Dateinamen des Ziels ohne das Verzeichnis. Wenn die zu erzeugenden Dateien voneinander unabhängig sind, sind rekursive Aufrufe problemlos.

Ein Anwendungsbeispiel dafür finden Sie in Abschnitt 23.3.1.

23.2.2 Ein Makefile für alles

Wenn es, wie oben erwähnt, um nur ein ausführbares Programm, zum Beispiel *projekt.exe*, geht, genügt ein einziges Makefile auf Projektebene. Makefiles in Unterverzeichnissen sind dann überflüssig. Hier wird wieder von der auf Seite 605 abgebildeten Verzeichnisstruktur ausgegangen. Das folgende Makefile geht vom gezeigten Verzeichnisbaum aus:

Listing 23.5: Makefile mit automatischer Ermittlung der cpp-Dateien

```
# cppbuch/k23/dirTree/makefiledep.mak   noch nicht optimal, siehe unten!
SRCDIR := src
verz := $(foreach dir,$(SRCDIR),$(wildcard $(dir)/*))
cppfiles := $(foreach dir,$(verz),$(wildcard $(dir)/*.cpp))
include include.mak
```

Mit Hilfe der Funktion `foreach` werden der Variablen `verz` die unterhalb *src* liegenden Verzeichnisse zugewiesen. Mit derselben Funktion werden alle cpp-Dateien dieser Verzeichnisse bestimmt. Zum Schluss wird die nachfolgende Datei *include.mak* eingelesen, die die restliche Verarbeitung übernimmt:

Listing 23.6: Include-Makefile

```
# cppbuch/k23/dirTree/include.mak
CXX := g++
CXXFLAGS := -g -Wall -c
INCLUDE := -I.
LDFLAGS := -g
OBJDIR := obj
BINDIR := bin
DISTDIR := dist
EXEFILE := projekt.exe
objects := $(subst $(SRCDIR), $(OBJDIR), $(cppfiles:.cpp=.o))
deps := $(objects:.o=.d)
.PHONY: all clean dist
all: $(BINDIR)/$(EXEFILE)
-include $(deps)
$(OBJDIR)/%.d: $(SRCDIR)/%.cpp
→mkdir -p $(@D)
→$(CXX) -MM -MT "$@ $(patsubst %.d,%.o,$@)" -MF $@ $<

$(OBJDIR)/%.o: $(SRCDIR)/%.cpp
→$(CXX) $(CXXFLAGS) $(INCLUDE) $< -o $@

$(BINDIR)/$(EXEFILE): $(objects)
→mkdir -p $(BINDIR)
→$(CXX) -o $@ $^ $(LDFLAGS)

$(DISTDIR)/gezippt.zip: $(BINDIR)/$(EXEFILE)
→mkdir -p $(DISTDIR)
→zip $(DISTDIR)/gezippt.zip $(BINDIR)/$(EXEFILE)

dist: $(DISTDIR)/gezippt.zip
```

```
clean:
→|$(RM) -r -f $(OBJDIR)
→|$(RM) -r -f $(BINDIR)
ultraclean:
→|$(RM) -r -f $(OBJDIR)
→|$(RM) -r -f $(DISTDIR)
→|$(RM) -r -f $(BINDIR)
```

include.mak enthält alle notwendigen Regeln. Im Vergleich zu den vorherigen Makefiles wurden die Ziele `dist` und `ultraclean` hinzugefügt. `make dist` sorgt dafür, dass die ausführbare Datei gezippt und im Verzeichnis *dist* abgelegt wird. Das letzte Ziel löscht sämtliche Verzeichnisse mit generierten Dateien. Einen kleinen »Schönheitsfehler« hat diese Lösung noch: Sie funktioniert, wenn es unterhalb von `$(SRCDIR)` genau noch eine Ebene von Verzeichnissen mit cpp-Dateien gibt, nicht aber, wenn es noch tiefer verschachtelte Verzeichnisse gibt. Dazu gibt es eine Lösung im nächsten Abschnitt.

23.3 Automatische Erzeugung von Makefiles

Leider ist es nicht möglich, mit *make* eine beliebig verschachtelte Struktur unterhalb `$(SRCDIR)` zu analysieren. Um das zu erreichen, hilft ein kleines Shell-Skript *makemakefile*, sodass nun auch dieses Problem gelöst ist:

Listing 23.7: Shell-Skript zur automatischen Makefile-Erzeugung

```
# cppbuch/k23/dirTree/makemakefile
SRCDIR=src
echo SRCDIR := $SRCDIR > makefile
echo cppfiles=\\ >> makefile
find $SRCDIR -name "*.cpp"\
  | sed "s%\.cpp%\.cpp\\\\%g" >> makefile
# Leerzeile
echo "" >> makefile
echo include include.mak >> makefile
```

Unter Unix muss das Skript mit `chmod u+x makemakefile` ausführbar gemacht werden.

Hinweis für Windows-Benutzer

Der Windows-Kommandointerpreter versteht das Skript nicht, weswegen ein kleiner Umweg gegangen werden muss. Es wird davon ausgegangen, dass MSYS und MinGW (siehe Seite 518) installiert und im Pfad sind. Zum Aufruf von *makemakefile* gehen Sie in das Verzeichnis, das *makemakefile* enthält, und tippen dort `sh makemakefile` ein. Das Shell-Programm *sh* kann mit dem Skript etwas anfangen, und das Makefile wird erzeugt.

Wenn *makemakefile* aufgerufen wird, geschieht Folgendes:

1. `SRCDIR := src` wird in die Datei *makefile* geschrieben.
2. `cppfiles=\` wird in die nächste Zeile geschrieben. Im Skript muss für den Backslash \\ geschrieben werden.
3. Der `find`-Befehl ermmittelt alle cpp-Dateien in allen Unterverzeichnissen, ausgehend vom Quellverzeichnis SRCDIR. Das Ergebnis wird dem Streameditor *sed* übergeben, der an jedes ».cpp« (das sich am Zeilenende befindet) einen Backslash anfügt. *sed* erfordert dazu \\\\. Die Wirkung eines Backslashs am Zeilenende ist, dass die Folgezeile als dazugehörig interpretiert wird, das Ergebnis also wie eine einzige Zeile wirkt.
4. Nach Anhängen einer Leerzeile wird die Zeile `include include.mak` in das Makefile geschrieben. Das Ergebnis ist die folgende Datei *makefile*.

```
SRCDIR := src
cppfiles=\
src/rational/rational.cpp\
src/appl/main.cpp\

include include.mak
```

Die Datei *include.mak* ist dieselbe wie oben. Der Aufruf `make` würde dann alle Abhängigkeiten aller cpp-Dateien in einem beliebig verschachtelten Verzeichnisbaum unterhalb `$(SRCDIR)` bestimmen, alle cpp-Dateien übersetzen und die Ergebnisse in einem entsprechenden Verzeichnisbaum unterhalb von `$(OBJDIR)` ablegen. Anschließend würde das ausführbare Programm *projekt.exe* im `$(BINDIR)`-Verzeichnis erzeugt werden. Ein abschließender Hinweis zu `$(SRCDIR)`: Durch den Ersetzungsmechanismus in *include.mak* bedingt darf `$(SRCDIR)` nicht leer sein oder nur auf das aktuelle Verzeichnis (.) verweisen.

23.3.1 Makefile für rekursive Aufrufe erzeugen

Im Verzeichnis *cppbuch* der Beispiele von der DVD finden Sie ein anderes kurzes Skript mit dem Namen *makemakefile*. Es erzeugt ein Makefile, in dem alle Makefiles der untergeordneten Verzeichnisse aufgerufen werden:

Listing 23.8: Shell-Skript zur Makefile-Erzeugung für rekursive make-Aufrufe

```
# cppbuch/makemakefile
rm -f makefile
echo all:> temporaer.txt
find . -name makefile\
  | sed "s%\./%\⟶|\$(MAKE) -C %g" \
  | sed "s%/makefile%%g" >> temporaer.txt
echo clean:>> temporaer.txt
find . -name makefile\
  | sed "s%\./%\⟶|\$(MAKE) -C %g" \
  | sed "s%/makefile% clean%g" >> temporaer.txt
mv temporaer.txt makefile
echo makefile erzeugt! Aufruf: make oder make clean
```

Der `find`-Befehl ermittelt alle Dateien mit dem Namen *makefile* in allen Unterverzeichnissen. Das Ergebnis wird dem Streameditor *sed* übergeben, der die Zeichenfolge »./« am Anfang jeder Zeile durch ein Tabulatorzeichen, gefolgt von »`$(MAKE) -C` «, ersetzt.

Die Zeichenfolge »/makefile« wird gelöscht, sodass nur der Verzeichnisname bleibt. Mit dem erzeugten Makefile können mit nur einem Befehl sämtliche Beispieldateien übersetzt werden. Dabei werden die Makefiles in den jeweiligen Verzeichnissen aufgerufen. Fast alle dieser Makefiles bestehen jeweils nur aus einer Zeile, die das passende Makefile aus dem Verzeichnis *cppbuch/make* einschließt. In den beiden Makefiles in *cppbuch/make* befindet sich vor `$(CXX)` ein Minuszeichen, damit der Vorgang bei Fehlermeldungen nicht abgebrochen wird. Im Übrigen kann man *make* beschleunigen, wenn es mit der Option `-jX` aufgerufen wird (X = Grad der Parallelität, z.B. 2 oder 4). Sie bewirkt, dass die Übersetzungen *parallel* ablaufen. Um tatsächlich eine signifikante Zeiteinsparung erreichen zu können, müssen natürlich Betriebssystem und Hardware Parallelität unterstützen und die Übersetzungsvorgänge unabhängig voneinander sein. Ein Nachteil der parallelen Abarbeitung ist jedoch, dass sich die Reihenfolge von Fehler- und anderen Meldungen ändern kann.

23.4 Erzeugen von Bibliotheken

Aus Abschnitt 3.3.2 wissen Sie, dass zusammengehörige Klassen und Funktionen zu Bibliotheksmodulen zusammengefasst werden können. In diesem Abschnitt geht es darum, wie man das macht und wie ein Bibliotheksmodul, auch kurz Bibliothek oder Library genannt, benutzt wird. Um die Darstellung möglichst einfach zu halten, soll in den folgenden Beispielen nur die Klasse `Rational` in ein Bibliotheksmodul transformiert werden. Es gibt zwei Projekte oder Sichtweisen:

1. Entwicklungsprojekt zum Erzeugen der Bibliothek. Das Ergebnis ist kein ausführbares Programm, sondern ein Bibliotheksmodul, das anschließend eingebunden werden kann (siehe 2.). Es ist typisch, dass Bibliotheksmodule nach ihrer Entwicklung nur selten geändert, aber oft benutzt werden. Die Verzeichnisstruktur ist, bezogen auf unser Beispiel:

```
libprojekt          // Projektverzeichnis
|-- lib             // Verzeichnis für das Bibliotheksmodul
|-- libinclude.mak  // wird von makefile inkludiert, s.u.
|-- makefile
|-- makemakefile    // von Seite 609 bekannt
'-- src
    '-- rational
        |-- rational.cpp
        '-- rational.h
```

2. Softwareprojekt, das ein oder mehrere Bibliotheksmodule nutzt. Verzeichnisstruktur, bezogen auf unser Beispiel:

```
anwendung           // Projektverzeichnis
|-- main.cpp
|-- makefile
'-- rational.h
```

rational.cpp fehlt. In *main.cpp* wird nur die Schnittstelle *rational.h* benutzt; die Implementierung der Funktionen findet sich im einzubindenden Bibliotheksmodul.

Es werden statische und dynamische Bibliotheksmodule unterschieden. Was damit gemeint ist und welche Vor- oder Nachteile damit verbunden sind, ist Inhalt der nächsten Abschnitte.

23.4.1 Statische Bibliotheksmodule

Statische Bibliotheksmodule werden, wie in Abbildung 3.9 (Seite 124) gezeigt, zu dem ausführbaren Programm gebunden (statisches Linken). Die so erzeugte Datei ist dementsprechend größer. *Vorteil:* Sie kann auf einen anderen Computer derselben Bauart und mit demselben Betriebssystemtyp kopiert werden und funktioniert dort wie auf dem Originalsystem. *Nachteil:* Wenn N Programme dieselbe statische Bibliothek benötigen, wird N mal der zugehörige Speicherplatz gebraucht, wenn die Programme gleichzeitig laufen.

Erzeugen eines statischen Bibliotheksmoduls

Ein statisches Bibliotheksmodul unterliegt in der GNU-Welt einer Namenskonvention. Der Name beginnt mit *lib*, danach folgt ein passender Name, und die Dateiendung ist *.a* (für Archiv). Das GNU-Programm *ar* erzeugt und modifiziert statische Bibliotheksmodule. Der entscheidende Auszug des für unser Beispiel geeigneten Makefiles sieht so aus:

Listing 23.9: Makefile zur Erzeugung einer statischen Bibliothek (Auszug)

```
# cppbuch/k23/staticLib/libprojekt/libinclude.mak (Auszug)
LIB := lib/librational.a

all: $(LIB)

$(LIB): $(objects)
→mkdir -p $(@D)
→ar cru $(LIB) $(objects)
```

Der Rest des Makefiles entspricht der Datei *include.mak* von Seite 608. Die Parameter `cru` bedeuten:

- `c` : Archiv ggf. ohne Warnung neu erzeugen (create).
- `r` : Ggf. vorhandene Dateien ersetzen (replace).
- `u` : Wie r, aber nur, wenn die Dateien neuer als die zu vorhandenen sind (update).

Einbinden eines statischen Bibliotheksmoduls

Die oben erzeugte Datei *lib/librational.a* wird eingebunden, indem sie dem Compiler beim Linken übergeben wird. Das Makefile zum Erzeugen unser schlichten Applikation:

Listing 23.10: Einbindung des Bibliotheksmoduls

```
# cppbuch/k23/staticLib/anwendung/makefile
.PHONY: clean

CXX := g++
CXXFLAGS := -c -g -Wall
LIBDIR := ../libprojekt/lib
```

```
LDFLAGS := -g -static -L$(LIBDIR) -lrational

projekt.exe: main.o
	$(CXX) -o projekt.exe main.o $(LDFLAGS)

main.o: main.cpp rational.h
	$(CXX) $(CXXFLAGS) main.cpp

clean:
	rm -f projekt.exe main.o
```

Auch hier ist eine Konvention zu beachten, wie an der Variablen `$LDFLAGS` zu sehen ist: Der Name des Archivs wird mit dem Schalter `-l` übergeben, aber der Anfang des Dateinamens *lib* und die Endung *.a* werden weggelassen. Der Schalter `-L` teilt dem Linker das Verzeichnis, in dem sich die Bibliotheksdatei befindet, mit. `-static` sorgt für die statische Einbindung.

23.4.2 Dynamische Bibliotheksmodule

Dynamische Bibliotheksmodule sind *nicht* in der ausführbaren Datei enthalten, sondern werden erst beim Start des Programms dazugebunden (dynamisches Linken). *Vorteil:* Wenn beliebig viele Programme dieselbe dynamische Bibliothek benötigen, wird der zugehörige Speicherplatz nur einmal gebraucht. *Nachteil:* Die ausführbare Datei kann auf einen anderen Computer derselben Bauart und mit demselben Betriebssystemtyp kopiert werden, funktioniert dort aber nur, wenn auch die dynamische Bibliothek mitgeliefert und an einer passenden Stelle installiert wird – andernfalls wird sie beim Start des Programms nicht gefunden.

Erzeugen eines dynamischen Bibliotheksmoduls

Auch für dynamische Bibliotheksmodule gilt die Konvention, dass ein Dateiname mit *lib* beginnt und die Endung weggelassen wird. Im Unterschied zu den statischen Bibliotheksmodulen ist die Endung *.so* (shared objects) unter Linux und *.dll* (dynamic link library) unter Windows. Das für unser Beispiel geeignete Makefile berücksichtigt in der Definition der Variablen `DYNLIB` das Betriebssystem, indem die Umgebungsvariable `OS` (operating system) daraufhin geprüft wird, ob sie die Zeichenkette »Windows« enthält. Die Funktion `findstring` gibt die gesuchte Zeichenkette zurück, sofern sie vorhanden ist, andernfalls wird der leere String zurückgegeben. Bei dieser Gelegenheit lernen Sie die gleich einen Bedingungsausdruck für *make* kennen. Mit `ifeq` wird das Ergebnis von `findstring` mit »Windows« verglichen. Die Variable `DYNLIB` wird in Abhängigkeit vom Ergebnis des Vergleichs definiert. Hier folgt nun das Makefile:

Listing 23.11: Makefile mit include zur Bibliothekserzeugung

```
# cppbuch/k23/dynLib/libprojekt/makefile
SRCDIR := src
cppfiles := src/rational/rational.cpp
include libinclude.mak
```

Die mit `include` eingebundene Datei enthält die Abfrage des Betriebssystems. Die dynamische Bibliothek wird mit der Option `-shared` erzeugt (Zeile vor dem `clean`-Target).

Listing 23.12: Include-Makefile mit Erzeugung einer dynamischen Bibliothek

```
# cppbuch/k23/dynLib/libprojekt/libinclude.mak
CXX := g++
CXXFLAGS := -g -Wall -c
INCLUDE := -I.
LDFLAGS := -g
OBJDIR := obj

ifeq "$(findstring Windows,$(OS))" "Windows"
DYNLIB := lib/librational.dll
else # Unix/Linux
DYNLIB := lib/librational.so
endif

objects := $(subst $(SRCDIR), $(OBJDIR), $(cppfiles:.cpp=.o))
deps := $(objects:.o=.d)

.PHONY: all clean dist

all: $(DYNLIB)

-include $(deps)

$(OBJDIR)/%.d: $(SRCDIR)/%.cpp
	mkdir -p $(@D)
	$(CXX) -MM -MG -MT "$@ $(patsubst %.d,%.o,$@)" -MF $@ $<

$(OBJDIR)/%.o: $(SRCDIR)/%.cpp
	$(CXX) $(CXXFLAGS) $(INCLUDE) $< -o $@

$(DYNLIB): $(objects)
	mkdir -p $(@D)
	$(CXX) -shared -o $(DYNLIB) $(objects)

clean:
	rm -r -f $(OBJDIR)
	rm -r -f $(DYNLIB)
```

Einbinden eines dynamischen Bibliotheksmoduls

Im Gegensatz zum statischen Binden wird in das ausführbare Programm beim Linken nur ein Verweis auf die Bibliothek integriert. Das Kommando zum Linken bleibt bis auf `-static` dasselbe wie zur Einbindung eines statischen Bibliotheksmoduls, zum Beispiel:

```
g++ -o projekt.exe main.o -Llibverzeichnis -lrational
```

Der Linker erkennt an der Endung der gefundenen Datei *.so* oder *.dll*, dass nur ein Verweis einzutragen ist. Das Linux-Programm *ldd* zeigt die eingetragenen dynamischen Bibliotheken. So ergibt der Aufruf `ldd projekt.exe`:

```
linux-gate.so.1 => (0xffffe000)
librational.so => not found
libstdc++.so.6 => /usr/local/lib/libstdc++.so.6 (0xb7eb0000)
```

```
libm.so.6 => /lib/libm.so.6 (0xb7e8a000)
libgcc_s.so.1 => /usr/local/lib/libgcc_s.so.1 (0xb7e7c000)
libc.so.6 => /lib/libc.so.6 (0xb7d4d000)
/lib/ld-linux.so.2 (0xb7fc3000)
```

Man sieht, dass der Verweis `librational.so` existiert, die Bibliothek aber nicht an den voreingestellten Plätzen zu finden ist. Spätestens bei Aufruf des Programms muss der Ort der Bibliothek bekannt sein.

Programmaufruf unter Linux

Unter Linux gibt es die Konvention, dass der Pfad zu Libraries, die nicht zum System gehören, in der Shell-Variablen `LD_LIBRARY_PATH` abgelegt wird. Dazu wird, bevor `projekt.exe` aufgerufen wird, in der Shell das Kommando

```
export LD_LIBRARY_PATH=../libprojekt/lib
```

eingegeben. Andernfalls würde der Aufruf von `projekt.exe` eine Fehlermeldung ergeben. Nachdem der Pfad zur Library angegeben worden ist, kann auch *ldd* die Bibliothek lokalisieren. Oben wird angenommen, dass sich das Verzeichnis *libprojekt* auf derselben Ebene wie das Verzeichnis, in dem `projekt.exe` ausgeführt wird, befindet. Im Allgemeinen ist das jedoch nicht der Fall, weswegen die zugehörigen Bibliotheken bei der Installation eines Programms in die entsprechenden Systemverzeichnisse für Libraries, zum Beispiel */usr/local/lib*, kopiert werden. Und wenn nicht, also `LD_LIBRARY_PATH` zur Geltung kommt, sollte nicht der relative, sondern der absolute Pfad eingetragen werden.

Programmaufruf unter Windows

Ein Aufruf des Programms `projekt.exe` unter Windows würde ohne weitere Maßnahmen auch zu einer Fehlermeldung führen. Windows sucht erst im aktuellen Verzeichnis nach der Bibliothek und danach im durch die Umgebungsvariable `PATH` definierten Pfad. Um eine DLL (dynamic link library) auffindbar zu machen, ergänzt man den Pfad durch Eingabe des Kommandos

```
PATH=%PATH%;..\libprojekt\lib
```

Oder man kopiert die DLL in eins der Verzeichnisse, die sich bereits im Pfad befinden, zum Beispiel *C:windows/system*. Das folgende Makefile hat ein Ziel `run`, das abfragt, ob das Bibliotheksverzeichnis im Pfad liegt (Windows) bzw. den `LD_LIBRARY_PATH` entsprechend einstellt (Linux).

Listing 23.13: Makefile mit Ziel run

```
# cppbuch/k23/dynLib/anwendung/makefile
.PHONY: run zeigeLibs clean

CXX := g++
CXXFLAGS := -c -g -Wall
LIBDIR := ../libprojekt/lib
LDFLAGS := -g -L$(LIBDIR) -lrational
EXE := projekt.exe
```

```
$(EXE): main.o
	$(CXX) -o $(EXE) main.o $(LDFLAGS)

main.o: main.cpp rational.h
	$(CXX) $(CXXFLAGS) main.cpp

zeigeLibs: $(EXE)
	ldd $(EXE)

run: $(EXE)
ifeq "$(findstring Windows,$(OS))" "Windows"
ifeq "$(findstring $(subst /,\, $(LIBDIR)),$(PATH))" ""
	@echo Keine Ausfuehrung! $(LIBDIR) muss im Pfad liegen!
else
	$(EXE)
endif
else # Unix/Linux
	export LD_LIBRARY_PATH=$(LIBDIR); ./$(EXE)
endif
clean:
	rm -f $(EXE) main.o
```


23.5 GNU Autotools

Es gibt große Projekte, deren ausführbare Programme auf vielen verschiedenen Computern und Betriebssystemen laufen sollen. Ein prominentes Beispiel dafür ist die GNU Compiler Collection (GCC). Die oben gezeigten Mechanismen wären für so eine komplexe Aufgabe nicht ausreichend. Es gibt dafür einen Satz von Werkzeugen, GNU Autotools genannt, die dabei helfen sollen. Besonders sind die Werkzeuge Autoconf und Automake zu nennen [GNU]. Hier wird aus folgenden Gründen nur kurz auf die Benutzung der Autotools eingegangen:

1. Die Autotools sind Standard bei GNU- und manchen anderen Open Source-Programmen. Die Kommandofolge
   ```
   ./configure
   make install
   ```
   zum Erzeugen eines Makefiles und anschließendem Übersetzen und Installieren eines Programms kennt fast jeder Linux-Benutzer. Punkt und Schrägstrich vor `configure` sind erforderlich, wenn das aktuelle Verzeichnis nicht im Pfad liegt.
2. Der Umfang der Möglichkeiten sprengt den Rahmen dieses Buchs.
3. Für viele Programme sind ist der in den vorherigen Abschnitten beschriebene Build-Prozess mit make ausreichend und komfortabel.
4. Der Einarbeitungsaufwand ist beträchtlich. Es muss die Syntax mehrerer Programme, die alle zusammenwirken, gelernt werden. Die Dokumentation ist für unerfahrene Entwickler kaum verständlich.

5. Die gewünschte Portabilität wird oft nicht oder nur mit erheblichem Anpassungsaufwand geleistet, besonders die Portierung von Linux auf Windows.

Die letzten beiden Punkte führten dazu, dass sich etliche Entwickler von den Autotools abgewendet haben und weiter abwenden. So hat sich unter anderem das Entwicklungsteam der Linux-Desktop-Oberfläche KDE für den Umstieg auf CMake entschieden [Neun] (*http://www.cmake.org/*). CMake wird im folgenden Abschnitt 23.6 kurz beschrieben. Für die Demonstration der Autotools sei die folgende Verzeichnisstruktur gegeben:

```
.                                Projektverzeichnis
|-- anwendung
|   '-- main.cpp
'-- libprojekt
    '-- src
        '-- rational
            |-- rational.cpp
            '-- rational.h
```

Ziel ist es, auf jeder Ebene geeignete Makefiles zu erzeugen:

- *Projektverzeichnis*: Das Makefile soll die Makefiles der Unterverzeichnisse aufrufen.
- *libprojekt*: Das Makefile soll eine statische Bibliothek *librational.a* erzeugen.
- *anwendung*: Das Makefile soll *main.cpp* übersetzen und dabei die Header-Datei *../libprojekt/src/rational/rational.h* einbinden. Die Objektdatei *main.o* soll mit der Bibliothek *librational.a* zur ausführbaren Datei *appl* zusammengebunden werden.

Dazu verlangen die Autotools in jedem der drei Verzeichnisse die Dateien *configure.ac*, *Makefile.am* zur Steuerung des Prozesses sowie die Dateien *AUTHORS*, *NEWS*, *ChangeLog* und *README*. Diese sechs Dateien müssen geschrieben werden, wobei der Inhalt der letzten vier beliebig ist – die Dateien müssen nur existieren. Weitere verlangte Dateien wie *COPYING*, *INSTALL* und einige Skripte können mit dem Befehl `automake -a` erzeugt werden. Um nun die *configure*-Dateien zu erzeugen, müssen *zuerst* in den Unterverzeichnissen und dann im Hauptverzeichnis die Kommandos

```
autoreconf
automake –a
```

aufgerufen werden. Anschließend können im Projektverzeichnis (oder selektiv in den Unterverzeichnissen) `./configure` und `make` aufgerufen werden. Im Verzeichnis *anwendung* findet sich anschließend die ausführbare Datei *appl*. Die Steuerungsdateien *configure.ac* und *Makefile.am* sind:

```
# cppbuch/k23/autotools/Makefile.am
SUBDIRS = libprojekt anwendung
```

```
# cppbuch/k23/autotools/configure.ac
AC_INIT(anw, 1.0, info@fehler.de)
AC_CONFIG_AUX_DIR([.])
AM_INIT_AUTOMAKE
AC_PROG_CXX
AC_CONFIG_FILES([Makefile])
AC_CONFIG_SUBDIRS([libprojekt anwendung])
AC_OUTPUT
```

Dabei steht AC für autoconf und AM für automake. Es gibt außer diesen beiden noch mehr Programme bzw. Skripte, die aber indirekt aufgerufen werden und hier nicht direkt sichtbar werden. Dem Makro `AC_INIT` wird der Name des Projekts, die Versionsnummer und eine E-Mail-Adresse für Fehlermeldungen mitgegeben. `AC_PROG_CXX` ermittelt den C++-Compiler. Zu den weiteren Makros bitte ich, [GNU] zu konsultieren. Im Projektverzeichnis wird letztlich auf die Unterverzeichnisse verwiesen, deren entsprechende Dateien folgen.

```
# cppbuch/k23/autotools/libprojekt/Makefile.am
noinst_LIBRARIES = librational.a
librational_a_SOURCES = src/rational/rational.cpp \
                        src/rational/rational.h
```

`noinst` heißt, dass die zu erzeugende Bibliothek nicht installiert, also in das entsprechende Systemverzeichnis (Voreinstellung */usr/local/lib*) kopiert werden soll. Die Quellen müssen angegeben und können nicht automatisch ermittelt werden.

```
# cppbuch/k23/autotools/libprojekt/configure.ac
AC_INIT([libprojekt],[1.1])
AC_CONFIG_AUX_DIR([.])
AM_INIT_AUTOMAKE
AC_PROG_CXX
AC_PROG_RANLIB
AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

`AC_PROG_RANLIB` sorgt für die Generierung einer Bibliotheksdatei.

```
# cppbuch/k23/autotools/anwendung/Makefile.am
AM_CXXFLAGS = -I../libprojekt/src/rational
bin_PROGRAMS = appl
appl_SOURCES = main.cpp
appl_LDADD = ../libprojekt/librational.a
```

Der Inhalt des Makros `AC_CXXFLAGS` wird dem Compiler übergeben, damit er *rational.h* findet. Der Name des zu erzeugenden Programms wird mit `bin_PROGRAMS` angegeben. Dieser Name wird dann als Präfix für die Angabe der Quelldateien und der hinzuzufügenden Library benutzt.

```
# cppbuch/k23/autotools/anwendung/configure.ac
AC_INIT(appl, 1.0, info@fehler.de)
AC_CONFIG_AUX_DIR([.])
AM_INIT_AUTOMAKE
AC_PROG_CXX
AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

23.6 CMake

Wie die Autotools erzeugt CMake Makefiles, die von *make* weiterverarbeitet werden. Nach meiner eigenen Erfahrung ist der Einarbeitungsaufwand im Vergleich zu den Autotools erheblich geringer. Es muss auch nur ein Bruchteil an Dateien produziert werden, nämlich nur eine kurze Datei namens *CMakeLists.txt* für jedes Verzeichnis! Zum Vergleich seien die Dateien für dieselbe kleine Aufgabe, die für die Autotools verwendet wurde, angegeben.

```
# cppbuch/k23/cmake/CMakeLists.txt
project (anw)
add_subdirectory (libprojekt)
add_subdirectory (anwendung)
```

```
# cppbuch/k23/cmake/libprojekt/CMakeLists.txt
add_library (rational STATIC src/rational/rational.cpp)
```

```
# cppbuch/k23/cmake/anwendung/CMakeLists.txt
include_directories(../libprojekt/src/rational)
link_directories(../libprojekt)
add_executable(appl main.cpp)
target_link_libraries(appl rational)
```

Das ist schon alles. Die Makefiles werden durch Aufruf im Projektverzeichnis (oder selektiv in den Unterverzeichnissen) erzeugt, indem nur

```
cmake .
```

eingegeben wird (Punkt nicht vergessen). Das ist alles! Damit hat *cmake* gute Chancen, die umständlichen Autotools abzulösen. Als Nachteil mag gesehen werden, dass die Installation von CMake vorausgesetzt wird, wenn auf einem System das Makefile neu konfiguriert werden soll, während eine von den Autotools erzeugte Distribution, die üblicherweise *configure* enthält, auf einem anderen System die Autotools nicht benötigt. Dieser minimale Nachteil wird verschwinden, wenn CMake es schafft, die Autotools abzulösen.

23.7 Code Bloat bei der Instanziierung von Templates vermeiden

In Projekten mit einer großen Anzahl von Dateien gibt es bei der Instanziierung von Templates das Phänomen der Aufblähung der Objektdateien (englisch *code bloat*). Zur Erklärung wird die Existenz der folgenden Template-Klasse angenommen:

Listing 23.14: Beispiel-Template

```
#ifndef MEINTYP_T
#define MEINTYP_T

template<typename T>
class MeinTyp {
   public:
     MeinTyp(T par) {
       // umfangreicher Code für den Konstruktor
     }

     T get() {
       // umfangreicher Code für get()
     }
   private:
     // Attribute weggelassen
};
#endif
```

Das Anwendungsprogramm bestehe aus 501 Dateien *main.cpp*, *prog001.cpp*, *prog002.cpp* usw. bis *prog500.cpp*. Es sei ferner angenommen, dass in jeder der *prog*-Dateien Objekte der Klassen `MeinTyp<string>` und `MeinTyp<double>` verwendet werden und deswegen jedesmal die Datei *meintyp.t* mit `#include` eingeschlossen wird. Dieses Compilationsmodell heißt deswegen auch Inklusions-Modell. Die Übersetzung aller cpp-Dateien liefert *main.o*, *prog001.o*, *prog002.o* usw. bis *prog500.o*. Diese Dateien enthalten alle den vollständigen Objektcode der Templateklasse, obwohl er in allen Dateien identisch ist (abgesehen vom Unterschied `string`/`double`)! Damit dauert erstens die Compilation relativ lange, weil der Objektcode jedesmal neu erzeugt wird, und zweitens wird eine Menge an Massenspeicherplatz verschwendet. Bei großen Projekten kann eine vollständige Compilation durchaus mehrere Stunden dauern. Der Linker hat die Aufgabe, die 498 überflüssigen Duplikate zu entfernen. Mit wenig Aufwand lässt sich das Verhalten entscheidend verbessern. Dazu gibt es einige Wege, die ein ungefähr gleich gutes Ergebnis liefern:

1. Verhindern der überflüssigen Objektcode-Erzeugung für Templates mit der Anweisung `extern template`.
2. Aufspaltung des Templates in Schnittstelle und Implementation

Die Methoden werden in den folgenden Abschnitten am Beispiel gezeigt. Die zweite Methode hat den Vorteil, dass der Compiler nicht jedesmal die ganze Implementation des Templates lesen muss, sondern nur genau einmal. Dafür hat sie den Nachteil, dass sie nur für selbst zu schreibende Templates anwendbar ist – die vorgegebenen Templates etwa der C++-Standardbiblothek können nicht verändert werden.

23.7.1 extern-Template

Eine Deklaration mit dem Schlüsselwort `extern` hat in Abschnitt 3.3.3 die Bedeutung, dass etwas benutzt werden kann, das anderweitig definiert ist. Hier wird es in einem ähnlichen Sinne verwendet. Zunächst sei das Hauptprogramm vorgestellt:

Listing 23.15: Hauptprogramm

```
// cppbuch/k23/templateInst/extern/main.cpp
#include"func01.h"
#include"func02.h"

int main() {
   func01();  // benutzt Template
   func02();  // benutzt Template
}
```

Im diesem Beispiel wird nicht von 500, sondern nur von zwei Dateien ausgegangen, die jeweils verschiedene Instanziierungen des Templates nutzen:

Listing 23.16: Erste Funktion

```
// cppbuch/k23/templateInst/extern/func01.cpp
#include"func01.h"
#include"meintyp.t"
#include<iostream>
#include<string>

// verhindert Instanziierung:
extern template class MeinTyp<std::string>;

void func01() {
   MeinTyp<std::string> mt("func01");
   std::cout << mt.get() << std::endl;
}
```

Listing 23.17: Zweite Funktion

```
// cppbuch/k23/templateInst/extern/func02.cpp
#include"func02.h"
#include"meintyp.t"
#include<iostream>
// verhindert Instanziierung:
extern template class MeinTyp<double>;

void func02() {
   MeinTyp<double> mt(3.1415926);
   std::cout << mt.get() << std::endl;
}
```

Auf die Darstellung der Header-Dateien wird verzichtet. Natürlich müssen die Templates wenigstens an einer Stelle instanziiert werden. Man fügt dem Projekt eine Datei, in der die Templates explizit instanziiert werden, hinzu:

Listing 23.18: Datei zur Instanziierung von Templates für zwei Datentypen

```
// cppbuch/k23/templateInst/extern/instanzen.cpp
#include"meintyp.t"
#include<string>
template class MeinTyp<std::string>;  // explizite Instanziierung
template class MeinTyp<double>;   // explizite Instanziierung
```

23.7.2 Aufspaltung in Schnittstelle und Implementation

Der Weg führt über die Aufspaltung der Datei *meintyp.t* in eine gleichnamige Datei, die ausschließlich die Prototypen enthält, und eine Datei *meintypImpl.t* mit der Implementierung:

Listing 23.19: Template-Prototypen

```
// cppbuch/k23/templateInst/trennIFimpl/meintyp.t
#ifndef MEINTYP_T
#define MEINTYP_T

template<typename T>
class MeinTyp {
   public:
     MeinTyp(T par);  // Prototyp
     T get();         // Prototyp
   private:
     // Attribute weggelassen
};
#endif
```

Listing 23.20: Template-Implementierung

```
// Implementierungsdatei cppbuch/k23/templateInst/trennIFimpl/meintypImpl.t
#include"meintyp.t"

template<typename T>
MeinTyp<T>::MeinTyp(T par) {
  // umfangreicher Code
}

template<typename T>
T MeinTyp<T>::get() {
  // umfangreicher Code
}
```

In den Dateien *func01.cpp* und *func02.cpp* kann die `extern`-Anweisung entfallen. Es fehlt nur die Datei, die die zu instanziierenden Typen enthält. Sie heiße ebenfalls *instanzen.cpp*, und *nur sie* benötigt die Implementierungsdatei. Die Datei *instanzen.cpp* wird dem Projekt hinzugefügt. Da sich nur in ihr der eigentliche Programmcode für das Template befindet, werden alle anderen cpp-Dateien erheblich schneller compiliert, der Platzbedarf für die Objektdateien schrumpft, und das Linken geht schneller.

Listing 23.21: Datei zur Template-Instanziierung

```
// cppbuch/k23/templateInst/trennIFimpl/instanzen.cpp
#include"meintypImpl.t" // Implementierung einlesen (nicht meintyp.t!)
#include<string>
template class MeinTyp<std::string>; // explizite Instanziierung
template class MeinTyp<double>;   // explizite Instanziierung
```

Wer mehr über die Template-Instanziierung (und überhaupt mehr über Templates) wissen möchte, dem sei [VaJo] empfohlen.