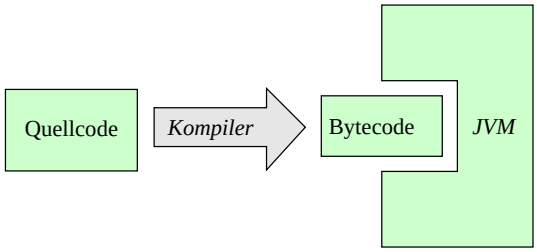
Kapitel 2

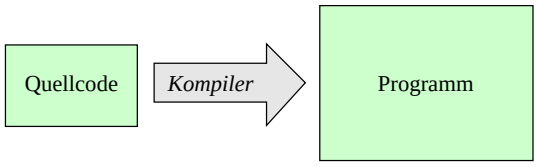
Kompilieren und Linken

Bevor wir uns auf C++ selbst stürzen, brauchen wir einiges Vorgeplänkel, wie man komfortabel ein größeres C++-Programm kompilieren kann. Mit Java stellt sich der Kompiliervorgang folgendermaßen dar:



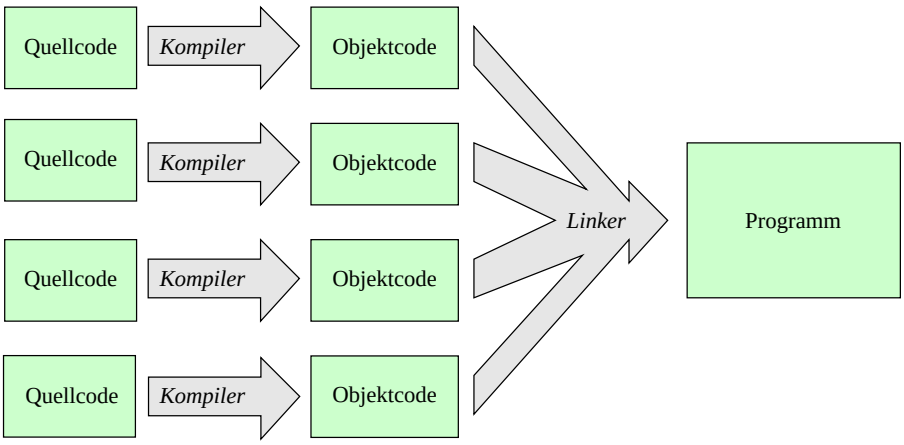
Der Java-Quellcode wird durch den Java-Compiler in Java-Bytecode übersetzt. Dieser wird von der Java Virtual Machine (JVM) ausgeführt. Ohne diese kann das Betriebssystem nicht viel mit dem Kompilat anfangen.

In C++ (wie auch in C, Fortran, Pascal) sieht es jedoch etwas anders aus:



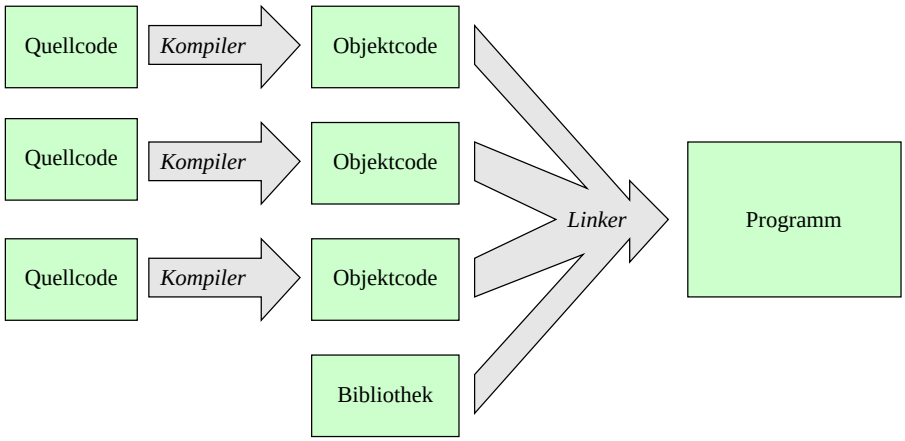
Das Endergebnis ist ein Programm, das direkt auf dem Betriebssystem aufsetzt. So etwas wie eine virtuelle Maschine ist zum Ausführen des Programmes nicht notwendig.

Dies war allerdings eine sehr grobe Sicht der Dinge. Wenn man es etwas genauer nimmt, dann stellt sich die Situation komplizierter dar:



Man kompiliert den C++-Code zuerst und erhält dadurch für jede Quelldatei eine Objektdatei. Diese Objektdateien werden anschließend vom so genannten Linker zum endgültigen Programm zusammengebunden.

Der erste Teil kann aus Programmierersicht in einem Teil der Fälle wegfallen:



Dann nämlich, wenn andere ihn bereits übernommen haben. In diesem Fall werden diese Teile als Bibliothek zur Verfügung gestellt. Man muss sie nur noch „dazulinken“.

2.1 Eigenschaften des Linkers

Was hat man sich also unter diesem ominösen „Linker“ vorzustellen:

- Der Linker ist ein *Entwicklungs*tool. Das bedeutet, dass der Anwender unseres Programms nichts mehr mit dem Linker zu tun hat.
- Der Linker ist ein extra Programm, das der Anwender unseres Programms nicht unbedingt hat. Der Anwender braucht es allerdings auch nicht.
- Der Linker stellt (im Allgemeinen) fest, ob alle Programmteile vorhanden sind. Wir können daher erwarten, eine Fehlermeldung zu bekommen, wenn wir einen Teil des Programms oder eine Bibliothek beim Linken vergessen. Eine Ausnahme ist das dynamische Linken, das in diesem Kurs aber nicht behandelt wird.

- Der Linker ist (im Allgemeinen) unabhängig vom Kompiler. Bis auf wenige Ausnahmen, geschieht der Linkerschritt unabhängig vom Kompilieren.[1]
- *Wir* müssen sicherstellen, dass die Objektfiles aktuell sind. Wenn man von obiger Ausnahme absieht, dann weiß der Linker nichts über das Kompilieren. Deshalb erscheint es konsequent, dass der Linker unabhängig davon geschieht. Es können schließlich auch Objektdateien zu einem Programm zusammengelinkt werden, die in unterschiedlichen Programmiersprachen geschrieben oder mit unterschiedlichen Kompilern erzeugt wurden.

2.2 Makefiles

Insbesondere der letzte Punkt, dass man sich als Programmierer selbst darum kümmern muss, ob die Objektdateien dem neuesten Stand der Quelldateien entsprechen, kann sehr lästig sein. Aber schließlich haben wir ja einen Computer, um solche Dinge zu automatisieren. Das Mittel der Wahl ist in diesem Fall ein so genanntes *Makefile*, welchen vom Programm *make* verarbeitet wird.

- Wir erstellen eine zentrale Datei, in der Regeln beschreiben, welche Objektdateien benötigt werden. Diese wird meist `Makefile` oder `makefile` genannt.
- *make* überprüft anhand der Zeit der letzten Dateiänderung, welche Quelltexte kompiliert und welche Objektdateien gelinkt werden müssen.
- Die Abhängigkeit können dabei beliebig kompliziert werden.

Das Format, in dem die Abhängigkeiten im Makefile beschrieben werden, ist folgendes:

```
Ziel: Abhängigkeiten
←TAB→Befehl
```

Am Beginn einer Zeile steht das Ziel des Schrittes, für den wir eine Regel angeben wollen. Nach einem Doppelpunkt folgen die Namen der Dateien, von denen diese Zieldatei abhängt, d.h. dass die Zieldatei neu erzeugt werden muss, wenn sich eine oder mehrere dieser Dateien geändert hat. In der darunterstehenden Zeile folgt zuerst ein Tabulator-Zeichen und danach der Befehl, wie man die Zieldatei erzeugt, also z.B. der Kompiler- oder Linkeraufruf.

Am besten sieht man jedoch an einem Beispiel, wie man dies benutzt:

```
hafas: hafas.o dijkstra.o
      g++ hafas.o dijkstra.o -Wall -g -o hafas
hafas.o: hafas.cc
      g++ -c -Wall -g hafas.cc -o hafas.o
dijkstra.o: dijkstra.cc
      g++ -c -Wall -g dijkstra.cc -o dijkstra.o
```

[1] Eine Ausnahme kann das generische Programmieren sein, wenn sich erst beim Linken herausstellt, dass noch Programmcode „generiert“ werden muss.

Für das Programm `hafas` benötigen wir die Objektdateien `hafas.o` und `dijkstra.o`. Zum Linken verwenden wir g++, dem wir die Namen der Objektdateien und als folgende Optionen mitgeben:

-Wall schaltet alle Warnungen ein. Selbstverständlich ist das nur sinnvoll, wenn man diese auch berücksichtigt. Eine gute Regel ist daher, seine Programme so zu schreiben, dass es keine Warnungen gibt.

-g schaltet die Generierung von Debugging-Informationen ein. Während der Entwicklungsphase kann man dadurch mit einem Debugger den Code während der Ausführung beobachten.

-o hafas gibt an, dass das Programm den Namen hafas bekommen soll. Gibt man diese Option nicht an, dann erzeugt es ein Programm a.out, was man sicherlich nicht haben möchte.

Die anderen beiden Regeln beschreiben, wie und wann die nötigen Objektdateien erzeugt werden. Erstaunlicherweise wird auch hierfür das Programm g++ verwendet.Das rührt daher, dass es letztendlich nur ein Frontend für Kompiler und Linker ist. Die Option -c gibt an, dass wir in diesem Fall den Quelltext kompilieren wollen.

Benutzt wird das Makefile z.B. mit dem Befehl `make hafas`, wenn das Programm hafas aktualisiert werden soll. Wenn nötig werden dazu `dijkstra.o` oder `hafas.o` neu erzeugt. Wenn `make` kein Argument übergeben wird, wendet es die erste Regel an. In diesem Fall würde also `make` das gleiche bewirken wie `make hafas`.

Mit Makefiles ist allerdings noch wesentlich mehr möglich. Im Rahmen dieser Einführung soll jedoch nur noch die Verwendung von Variablen angekratzt werden. Hierfür erweitern wir unser Beispiel:

```
CC = g++
FLAGS = -Wall -g
hafas.o: hafas.cc
      $(CC) -c hafas.cc $(FLAGS) -o hafas.o
dijkstra.o: dijkstra.cc
      $(CC) -c dijkstra.cc $(FLAGS) -o dijkstra.o
hafas: hafas.o
      $(CC) hafas.o dijkstra.o $(FLAGS) -o hafas
```

In diesem Fall wird am Anfang der Datei der Name des Kompilers bzw. die Optionen in einer Variablen gespeichert. Den Wert einer Variablen erhält man in einem Makefile, wenn man den Namen in Klammern setzt und ein Dollarzeichen davorschreibt. Im obigen Fall hat die Verwendung einer Variable den offensichtlichen Vorteil, dass man den Kompiler oder die Optionen an einer zentralen Stelle für alle Aufrufe ändern kann.

Als weiterführende Literatur zu Makefiles empfehle ich: Oram/Talbott *Managing Projects with make*. O'Reilly & Associates, Inc.

2.3 Ein unerwartetes Beispiel

Um die Verwendung von Makefiles etwas interessanter zu machen, stellen wir hier noch ein kleines Beispiel aus einem anderen Kontext vor:

```
diplomarbeit.ps: diplomarbeit.dvi bild.eps
      dvips diplomarbeit -o
diplomarbeit.dvi: diplomarbeit.tex bild.eps
      latex diplomarbeit
bild.eps: bild.jpg
      jpeg2ps bild.jpg > bild.eps
```

Es geht darum, die lästigen Aufrufe von LaTeX und dvips beim Erstellen eines Textes zu automatisieren. Außerdem soll wenn nötig eine JPEG-Datei, die im Dokument verwendet wird, in Postscript umgewandelt werden.

2.4 Aufgaben

1. Analysieren Sie das obige Makefile! Was passiert, wenn man diplomarbeit.tex ändert und dann make aufruft? Was passiert, wenn man bild.jpg ändert und dann make aufruft?

2. Schreiben Sie ein Makefile für eines Ihrer Projekte, bei dem mindestens zwei Regeln voneinander abhängig sind. (Ob es sich bei ihrem Projekt um ein Programm, ein Textdokument mit Latex oder sonst irgendwas handelt ist zweitrangig.)