Kapitel 3

Headerfiles

3.1 Eine erste Klasse

Der im vorherigen Kapitel vorgestellte Mechanismus mit Makefile ist schön und gut, aber ziemlich langweilig, wenn man ihn nicht mit Leben füllt. Es ist daher höchste Zeit, dass wir ein paar erste Schritte in C++ machen und uns eine erste Klasse anschauen. Auf den ersten Blick erinnert die Syntax (nicht von ungefähr) der von Java:

```
class Vektor {
private:
  double myX,myY;
public:
  Vektor(double x, double y):myX(x),myY(y) {};

Vektor addiere(Vektor v)
  { return Vektor(myX+v.myX,myY+v.myY); }
};
```

Es wird eine Klasse Vektor definiert, die einen zweidimensionalen Vektor repräsentieren soll. Beim Konstruktor wird der aufmerksame Leser sicherlich einen ersten Unterschied feststellen. Es ist in C++ möglich, einen Konstruktor einer Variable der Klasse aufzurufen – ähnlich dem Konstruktor der Basisklasse. In diesem Fall wird myX mit x initialisiert. Wichtig ist dabei, dass die Variablen in der Reihenfolge initialisiert werden, wie sie deklariert wurden. Leicht vergessen wird bei der Klassendeklaration das Semikolon, das nach der Klasse stehen muss.

Wenn man keinen Konstruktor definiert, hat eine Klasse automatisch einen Standardkonstruktor. Dieser hat keine Parameter und ruft nur die Standardkonstruktoren der Datenelemente dieser Klasse auf. (Im obigen Fall geschieht dies also für `myX` und `myY`.) Wenn man jedoch einen (anderen) Konstruktor definiert, so wird *kein* Standardkonstruktor definiert. Man muss dies gegebenenfalls selbst machen. Im konkreten Fall ist es daher nicht möglich, eine Instanz von `Vektor` zu erzeugen, die nicht initialisiert wird.

3.2 Deklaration und Definition

Die Funktion `addiere` soll uns nun als Beispiel dienen, um die Begriffe *Deklaration* und *Definition* einzuführen. In C++ ist es nämlich so, dass dem Benutzer einer Funktion nicht bekannt sein muss, wie diese in Wirklichkeit aussieht. Trotzdem wird überprüft, ob die Funktion vorhanden ist. Man benötigt auch keine Interfaces (die es in C++ in dieser Form gar nicht gibt), sondern schreibt einfach den Kopf der Funktion:

```
class Vektor {
private:
  double myX,myY;
public:
  Vektor(double x, double y);
  Vektor addiere(Vektor v);
};
```

Nachdem man die Funktion *deklariert* hat, muss sie natürlich an anderer Stelle im Code noch *definiert* werden.

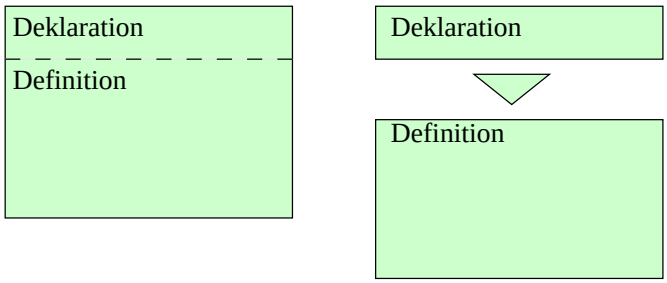
```
Vektor::Vektor(double x, double y)
    :myX(x),myY(y) {}

Vektor Vektor::addiere(Vektor v)
  { return Vektor(myX+v.myX,myY+v.myY); }
```

Da dies außerhalb der Klasse `Vektor` passiert, muss man angeben, dass diese Funktion zu dieser Klasse gehört. Dies geschieht wie oben gezeigt, indem man den Klassennamen getrennt durch zwei Doppelpunkte vor den Namen der Funktion schreibt.

Was nun allerdings noch fehlt, ist die Motivation, warum man dies machen sollte. Dahinter stehen folgende Überlegungen:

- Die Definition also den Funktionskörper zu Verarbeiten dauert (relativ) lange.
- Zum Kompilieren der anderen Programmteile, die diese Funktion benutzen, wird nur die Deklaration benötigt.
- Es soll aber ein Mechanismus kreiert werden, der überprüft ob es diese Funktion überhaupt gibt.

Was man daher üblicherweise macht, ist die Teile zu trennen:



In unserem konkreten Beispiel sieht das dann so aus:

vektor.h

```
class Vektor {
private:
  double myX,myY;
public:
  Vektor(double x, double y);

  Vektor addiere(Vektor v);
};
```

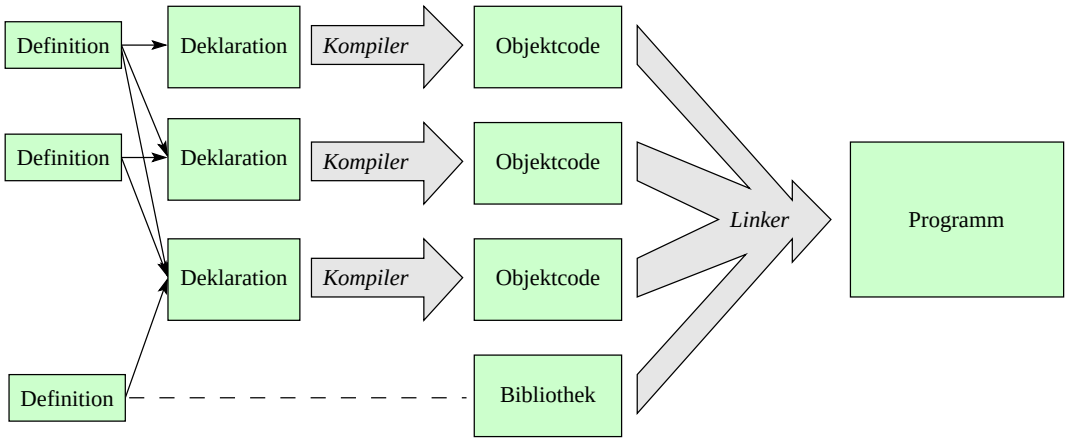
Das „h“ steht für „head“, da die Datei von den Funktionen nur den Funktionskopf enthält. Man nennt sie denglisch auch Headerfiles. Der Funktionsrumpf kommt dann in eine separate Datei:

vektor.cc

```
#include "vektor.h"
Vektor::Vektor(double x, double y)
    :myX(x),myY(y) {}

Vektor Vektor::addiere(Vektor v)
  { return Vektor(myX+v.myX,myY+v.myY); }
```

Die erste Zeile bewirkt dabei, dass der Inhalt von vektor.h an dieser Stelle eingelesen wird. Der Effekt ist derselbe, als wenn man den Inhalt von vektor.h an die Stelle geschrieben hätte. Dadurch ist dem Kompiler, wenn er die Definition des Konstruktors oder der Funktion `Vektor::addiere` verarbeitet, die Deklaration von Vektor bekannt. Insbesondere kann er – und tut auch – überprüfen, ob tatsächlich ein Konstruktor mit zwei `double`s vorgesehen ist und `addiere` eine Funktion in der Klasse `Vektor` ist.

Die typische Struktur eines Programms in C++ (oder C, Fortran, Pascal) ist damit nochmal genauer:



3.3 Makefile – mit Header-Files

Inzwischen sieht man schon, dass bei der Verwendung von Headerfiles die Abhängigkeiten reichlich unübersichtlich werden können. Schließlich muss man eine Datei neu kompilieren, wenn sich eine Header-Datei ändert, die benutzt wird. Es könnte ja sein, dass es eine verwendete Funktion aus dieser Header-Datei nicht mehr gibt. Aber zum Glück haben wir ja die Möglichkeit, das mit unserem Makefile erledigen zu lassen:

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

3.4 Compiler-Guards

Der Kompiler überprüft, ob eine Klasse (Funktion, Variable) schon einmal deklariert wurde. Wenn man ein Header-File mehrfach einbindet, führt dies zu einer Fehler-Meldung, was wir verhindern müssen. Von „Hand“ ist das insbesondere bei Standard-Files mühsam, die an allen möglichen Stellen eingebunden werden. Daher gibt es auch hierfür einen verbreiteten Automatismus: Wir bauen eine Abfrage ein, ob diese Datei schon eingelesen wurde. Das geschieht derart, dass eine Markierung definiert wird, wenn die Datei bearbeiten wird. Falls die Datei noch einmal eingelesen wird, wir die Markierung erkannt und die Datei ignoriert. Ganz konkret sieht das so aus:

vektor.h
```
#ifndef VEKTOR_INC
#define VEKTOR_INC
class Vektor {
private:
  double myX,myY;
public:
  Vektor(double x, double y)
    :myX(x),myY(y) {};

  Vektor addiere(Vektor v);
};
#endif
```

In der ersten Zeile wird abgefragt, ob das so genannte Makro `VEKTOR_INC` bereits definiert ist (was es standardmäßig nicht ist). Falls nicht, wird das Makro definiert und der Rest der Datei bis zum `#endif` vom Kompiler verarbeitet. Falls nun diese Datei beim gleichen Kompiliervorgang noch einmal eingelesen wird, ist das Makro `VEKTOR_INC` bereits definiert und dieser Teil wird übersprungen.

An der Definition müssen wir dazu nichts ändern:

vektor.cc
```
#include "vektor.h"
Vektor Vektor::addiere(Vektor v)
  {
    return Vektor(myX+v.myX,myY+v.myY);
  }
```

3.5 Zusammenfassung von Kompiler-Direktiven

Ganz nebenbei haben wir nun ein paar Kompiler-Direktiven kennengelernt, das heißt Anweisungen, die nicht Code erzeugen, sondern beeinflussen, was der Kompiler macht. Hier noch eine kleine Zusammenfassung derselben und ihrer Bedeutung:

`#include "Dateiname"`
: Platziere den Inhalt der Datei an diese Stelle.

`#include <Dateiname>`
: Platziere den Inhalt der Datei an diese Stelle; Variante für System-Dateien (Ein- und Ausgabe, Container-Klassen, mathematische Funktionen)..

`#ifndef MACRO`
: Bearbeite nachfolgenden Code nur, wenn MACRO nicht definiert ist.

`#endif`
: Ende der Bedingung

`#define MACRO`
: Definiere MACRO

Es gibt noch einige Direktiven mehr. Bis auf wenige Ausnahmen, sollte die Verwendung von Kompiler-Direktiven jedoch vermieden werden! Die prominentesten dieser Ausnahmen, sind die obigen beiden. Für quasi alle anderen Anwendungen, bei denen man in C noch Kompiler-Direktiven verwendet hat, gibt es in C++ bessere Lösungen.

3.6 Ausgabe

Zu guter letzt soll noch das nötige Handwerkszeug vervollständigt werden, damit endlich ein erstes sinnvolles C++-Programm geschrieben werden kann. Dafür ist es sinnvoll, wenn man etwas auf der Konsole ausgeben kann. Dies ist durch die Verwendung einer System-Datei möglich:

ausgabe.cc

```
#include <iostream>
int main()
{
  int a(5);
  int b(a+3);
  std::cout << ¨a ist ¨ << a
    << ¨ und b ist ¨ << b << std::endl;
  return 0;
}
```

Es wird das Headerfile `iostream` eingelesen, welches die benötigten Klassen bereitstellt. Da es sich um eine Systemdatei handelt, wird der Dateiname nicht in Hochkommata sondern in spitzen Klammern angegeben.

Als nächstes kommt die Funktion `main`, die beim Aufruf des Programms ausgeführt wird. Beachten Sie, dass diese *nicht* in einer Klasse steht, sondern einfach so nackt im Raum. (Das ist in C++ generell für Funktionen möglich, aber nicht besonders objektorientiert.) Zuerst werden zwei Variablen `a` und `b` deklariert und initialisiert. Man beachte, dass die Initialisierung im Unterschied zu Java hier ohne `new` geschieht. Anschließend werden sie mit einigem Text ausgegeben. `std::cout` steht für console out, d.h. es soll auf die Konsole ausgegeben werden. Allem, was dort rausgeschickt wird, wird ein `<<` vorangestellt. Das Argument `std::endl` hat dabei eine Sonderrolle. Es bewirkt zum einen einen Zeilenvorschub, zum anderen dass der Puffer bei der Ausgabe geleert wird, d.h. dass der ausgegebene Text tatsächlich nun angezeigt wird.

Für eine Erklärung, wie der Mechanismus `<<` funktioniert, muss an dieser Stelle leider auf Kapitel 8.6 vertröstet werden.

Zum Schluss soll noch auf einen häufigen Fehler von Java-Umsteigern aufmerksam gemacht werden, der im Zusammenhang mit Konstruktoren auftritt:

kontostand.cc
```
int main ()
{
  double kontostand();
  kontostand = 100.0;
}
```

Wenn man versucht das zu Kompilieren, erhält man folgende Fehlermeldung:

```
pluto07: ~/lehre/prad_W03 >  g++ kontostand.cc
kontostand.cc: In function ‘int main()’:
kontostand.cc:6: assignment of function ‘kontostand()’
kontostand.cc:6: assignment to ‘double ()()’ from ‘double’
```

Die kryptische Fehlermeldung hilft an dieser Stelle nicht sonderlich weit. Es kann daher Stunden dauern, bis man den Fehler findet. Es geht aber wahrscheinlich schneller, wenn man sich daran erinnert hier gelesen zu haben, dass die Klammern „()“ nach dem Kontostand in C++ *nicht* stehen dürfen. Zur Erklärung nur soviel: Schuld daran ist ein Erbe aus C, so dass diese Konstruktion anders interpretiert wird, als wir es gerne hätten. (Da wir aber auch ohne dieses Erbe – nämlich Zeiger auf Funktionen – auskommen können, werden sie in diesem Text nicht weiter thematisiert.)

3.7 Aufgaben

1. Kompilieren Sie das Programm ausgabe.cc und rufen es auf.
2. Erweitern Sie die Klasse `vektor` um eine Funktion `ausgabe`, die den Vektor auf der Konsole ausgibt.
3. Verwenden Sie die Klasse in ihrem Programm, indem Sie einen Vektor definieren, einen zweiten dazuaddieren und das Ergebnis ausgeben.
4. Schreiben Sie eine Klasse `komplex`, die eine komplexe Zahl darstellt. Schreiben Sie Funktionen für Addition, Subtraktion, Multiplikation und Division sowie eine Ausgaberoutine. Testen Sie ihre Klasse mit einem kleinen Testprogramm.