Von Java nach C++

Thomas Willhalm

16. November 2004

Inhaltsverzeichnis

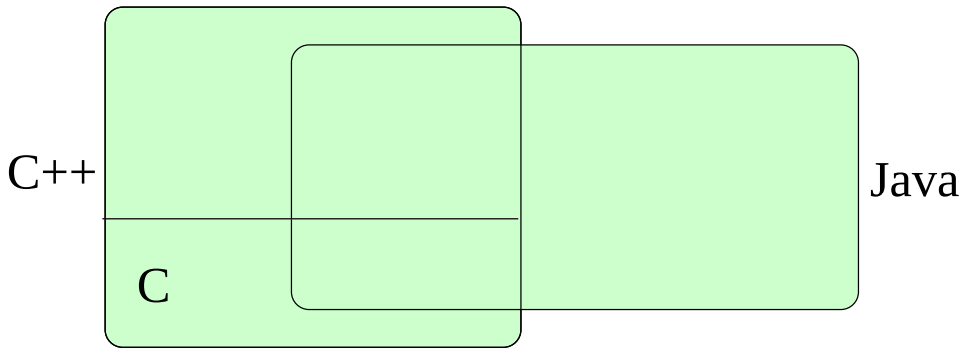
Kapitel 1

Einleitung

1.1 Vorwort

Dieser Text entstand am Lehrstuhl Wagner im Rahmen der Praktika, die Studierende im Hauptstudium machen müssen. Sie lernen im Grundstudium objektorientiertes Programmieren anhand der Programmiersprache *Java*. Es hat sich allerdings gezeigt, dass für Algorithmen und Datenstrukturen Java schlechter geeignet ist als C++. Deshalb sind wir dazu übergegangen, im ersten Teil des Praktikums C++ zu vermitteln. Dabei beschränken wir uns auf einige Teilaspekte von C++, die wir für notwendig und sinnvoll für diese Awendung erachten. Insbesondere werden hardwarenahe Teile der Programmiersprache auf eine Minimum reduziert.

Es soll aber nicht gesagt werden, dass Java eine schlechte Programmiersprache wäre oder gar gänzlich unnütz. Es ist vielmehr so, dass wir Dinge, die Java bereitstellt, nicht aber dafür andere Dinge, die es nicht bereitstellt, sehr wohl benötigen. Aus unserer Sicht der Dinge stellt sich das Verhältnis von Java, C und C++ folgendermaßen dar:



C++ kann seine Herkunft von C schon allein durch den Namen sicherlich nicht leugnen. Die meisten Teile von C sind in C++ enthalten. Es bietet aber durch die Mittel zum objektorientierten und generischen Programmieren weit mehr. In Java findet man (derzeit) nur die Hilfsmittel zur Objektorientierung wieder. Es setzt sich aber insbesondere bei der Speicherverwaltung stark von C und C++ ab. Auf der anderen Seite gehört eine wesentlich umfangreichere Bibliothek zu Java dazu, die z.B. die Programmierung von GUIs oder Netzwerkfähigkeit ermöglicht. Diese Dinge spielen jedoch bei Algorithmen und Datenstrukturen, im Gegensatz zur Speicherverwaltung, eine untergeordnete Rolle. Letztere ist dem Programmierer in Java jedoch gänzlich aus den Händen genommen. Darüberhinaus gibt es viele weitere Details,

die es dem Programmierer ermöglichen, effizienteren Code zu schreiben. Summa summarum erscheint uns daher die Verwendung von C++ für unseren Zweck sinnvoll.

1.2 Voraussetzungen

Voraussetzung für diesen Kurs sind wie gesagt Kenntnisse der Programmiersprache Java. Insbesondere sollten allgemeine Programmierkonzepte wie

- Variablen
- Schleifen und Verzweigungen
- Unterprogramme
- Objektorientierung
- Syntax

bekannt sein. Darüberhinaus ist praktische Erfahrung beim Programmieren wie das Interpretieren von Fehlermeldungen hilfreich.

1.3 Was C++ gegenüber Java fehlt

Der geübte Java-Programmierer wird bei einem Wechsel zu C++ sicherlich einige Dinge vermissen:

- Grafik (incl. GUI)
- Threads, Prozesssteuerung
- Netz
- Hilfen zur Dokumentation
- Reflection
- Garbage Collection ($\Rightarrow$ Speicher selbst verwalten)

Selbstverständlich ist es möglich, diese Defizite durch zusätzliche Programme oder Programmbibliotheken auszugleichen. In diesem Text wird darauf allerdings nicht eingegangen werden, da sie in unseren Praktika nicht nötig waren.

1.4 Danksagung

An dieser Stelle möchten wir Gabi Dorfmüller und Thomas Puppe danken, die als Versuchskaninchen für diese Einführung hergehalten haben und wertvolle Hinweise auf Lücken und Unzulänglichkeiten aufgedeckt haben. Außerdem danke ich Sabine Cornelsen für das Korrekturlesen.

1.5 Literatur

Dieser Text kann und soll beim Erlernen von C++ nur als Leitfaden dienen. Daher ist es wahrscheinlich notwendig, aber sicherlich sinnvoll, einige alternative Informationsquellen zu benutzen. Exemplarisch möchte ich folgende nennen:

- *ISO/IEC 14882 - Programming languages - C++*

- Stroustrup, Bjarne:
  The C++ programming language
  Addison-Wesley

- *The C++ report/Journal of Object-Oriented Programming.*
  SIGS Publ.

- Newsgroup *comp.lang.c++.moderated*

- John Lakos:
  Large scale C++ software design
  Addison-Wesley

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides:
  Design Patterns, Elements of Reusable Object-Oriented Software
  Addison-Wesley

- John J. Barton; Lee R. Nackman:
  Scientific and engineering C++
  Addison-Wesley

- Scott Meyers:
  Effective C++ und *More effective C++*
  Addison-Wesley-Longman

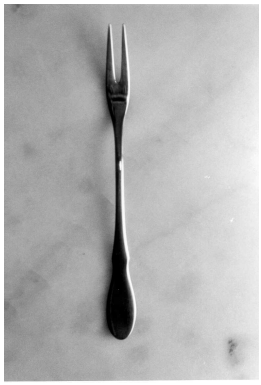
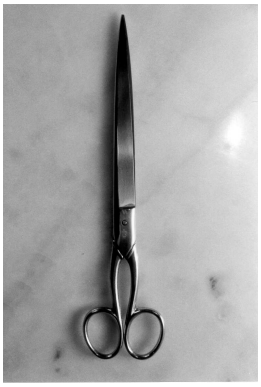
- Krzysztof Czarnecki; Ulrich Eisenecker:
  Generative programming : methods, tools, and applications
  Addison-Wesley

- Andrei Alexandrescu:
  Modern C++ Design : generic programming and design patterns applied
  Addison-Wesley

- Herb Sutter:
  Exceptional C++ und *More Exceptional C++*
  Addison-Wesley

1.6 Warnung

Da es ja in diesem Text darum geht, einige Teile von C++ vorzustellen, die in Java „weggelassen“ wurden, möchten wir eine Warnung vorweg schicken. Es ist nämlich nicht so, dass die Entwickler von Java keinen Grund gehabt hätten, dass sie nicht alle Konstruktionen von C++ übernommen haben. Zumindest in einigen Fällen war ihre Motivation, den Programmierer vor sich selbst zu schützen. Es gibt durchaus Gefahren, die durch die neuen Möglichkeiten entstehen, und wir sind uns dieser Gefahren bewusst. Mit den neuen Werkzeugen von C++ verhält es sich aber unserer Meinung nach ähnlich wie mit jenen aus dem bekannten Kindervers:

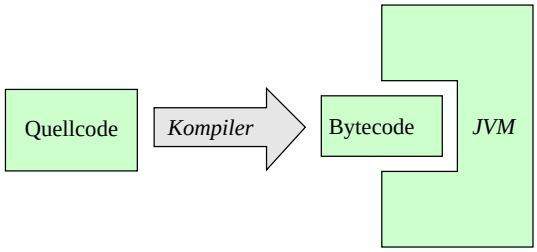

Messer, Gabel, Schere, Licht, ist für kleine Kinder nicht!

In beiden Fällen können die genannten Werkzeuge aber ziemlich hilfreich sein. (Oder haben Sie schon einmal versucht, ein Steak ohne Messer und Gabel zu essen?) Letztenendes ist unsere Hoffnung, dass sich der Leser als erwachsen genug erweisen wird, mit der dunklen Seite von C++ fertig zu werden.

Kapitel 2

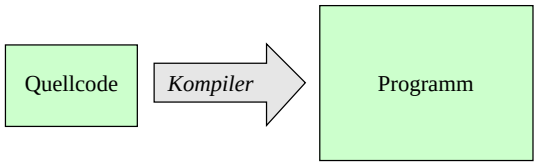
Kompilieren und Linken

Bevor wir uns auf C++ selbst stürzen, brauchen wir etwas Vorbereitung, um komfortabel größere C++-Programme kompilieren zu können. Mit Java stellt sich der Kompiliervorgang folgendermaßen dar:



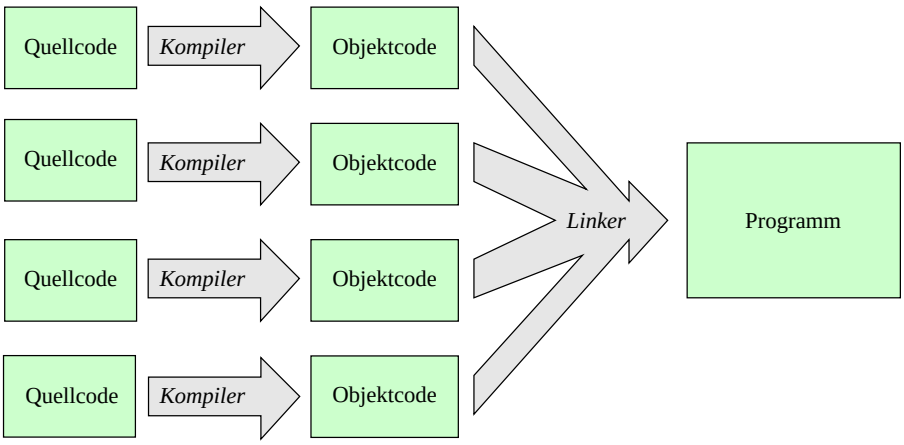
Der Java-Quellcode wird durch den Java-Compiler in Java-Bytecode übersetzt. Dieser wird von der Java Virtual Machine (JVM) ausgeführt. Ohne diese kann das Betriebssystem nicht viel mit dem Kompilat anfangen.

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



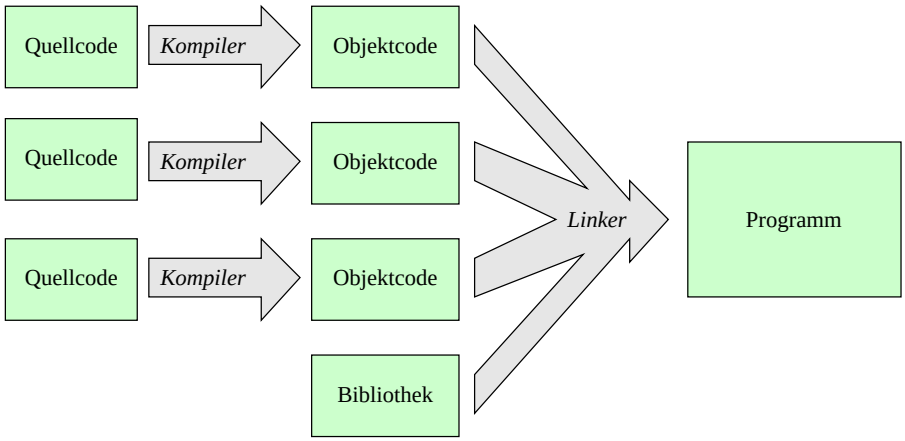
Das Endergebnis ist ein Programm, das direkt auf dem Betriebssystem aufsetzt. So etwas wie eine virtuelle Maschine ist zum Ausführen des Programmes nicht notwendig.

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.)

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.

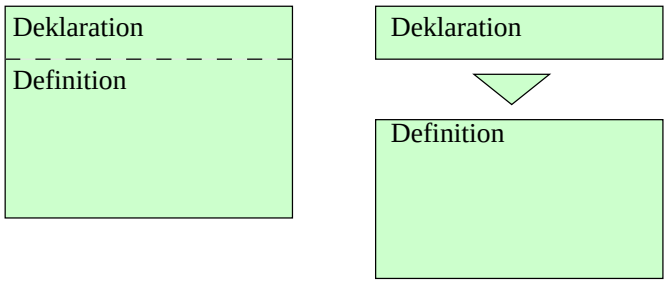
```
Vektor::Vektor(double x, double y)
    :myX(x),myY(y) {}

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);
};
```

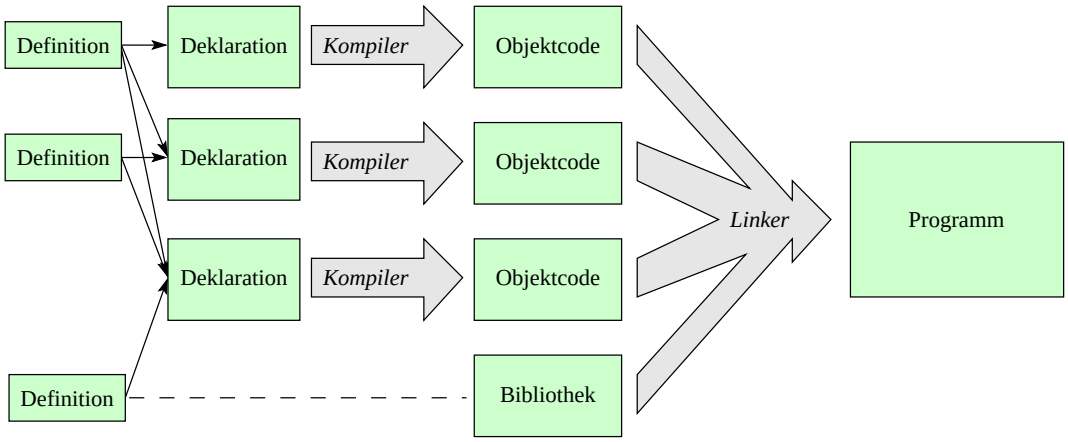
Das „h“ steht für „head“, da die Datei von den Funktionen nur den Funktionskopf enthält. Man nennt sie denglisch auch Headerdatei. Der Funktionsrumpf kommt dann in eine separate Datei:

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 (und tut) er 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 Header-Files 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-Headerfiles 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, wird 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 wirklich weiter. 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.

Kapitel 4

Qualifier

4.1 Qualifier static und extern

static	extern
Nur den Programmteilen bekannt, die die Definition sehen.	Allen Programmteilen bekannt, die dazugelinkt werden.
Standard für Variablen	Standard für Funktionen
`static int abs(int i);`	`extern node_array xpos;`
besser: private Funktion einer Klasse	besser: als Parameter Übergeben

Tabelle 4.1: static und extern in C/C++

So ähnlich wie es private- und public-Funktionen gibt, die außerhalb der Klasse unsichtbar bzw. sichtbar sind, so kann man auch Funktionen unterscheiden, die außerhalb der Quelldatei oder genauer gesagt der Objektdatei unsichtbar bzw. sichtbar sind. Beim Linken ist es also nicht möglich, die „unsichtbaren“ Funktionen zu verwenden. Unglücklicherweise nennt man diese dann aber nicht `private` und `public` sondern `static` und `extern`. Gesagtes gilt übrigens mutatis mutandis für Variablen. Eine Übersicht bietet Tabelle 4.1.

Wie bei vielen Dingen, ist dies ein Relikt von C, für die es in C++ meist die bessere Alternative der Kapselung durch Objekte gibt. Statische Funktionen sind daher nahezu überflüssig geworden. Wenn eine Klasse aber nur aus einer Funktion bestünde, dann erscheint es wenig sinnvoll, diese in eine Klasse zu packen. Statische Variablen sortiert man aber besser zu den zugehörigen Funktionen in eine Klasse. Externe Variablen sind allerdings ganz schlechter Stil. Man sollte ihren Inhalt besser als Parameter übergeben.

4.2 static – eine andere Bedeutung

Besonders unglücklich ist die Namensgebung für `static` in Abschnitt 4.1, da es auch noch eine weitere Bedeutung hat, die von Java bereits bekannt sein sollte:

Beispiel:

```
class Vektor {
    ⋮
  static int maximalLength;
  static Vektor generateRandom();
    ⋮
}
```

So eine Variable existiert nur einmal pro Klasse (und nicht pro Instanz) und so eine Funktion ist unabhängig von der Instanz, d.h. sie kann auch nur statische Variablen und andere statische Funktionen benutzen.

4.3 Qualifier inline

Zusätzlich zum Erbe `static` und `extern` aus C gibt es in C++ noch einen Qualifier, der sich aber als sehr nützlich erweist. Die Rede ist von `inline`:

Beispiel

```
inline int abs(int i)
{ return i<0 ? -i : i; }
```

`inline` ist ein Hinweis an den Kompiler, den Code direkt einzufügen anstatt einen wirklichen Funktionsaufruf zu machen. Das spart einen Funktionsaufruf (welcher hier verhältnismäßig viel Zeit kostet), vergrößert aber eventuell das Programm. Selbstverständlich muss das Verhalten ein Funktion das gleiche sein, wenn sie Funktion aufgerufen wird, egal ob sie `inline` deklariert wird oder nicht. Der Kompiler muss sich allerdings nicht an den Hinweis halten, dass wir diese Funktion gerne `inline` hätten. Was er tut werden wir aber nie erfahren, wenn wir nicht in den erzeugten Assembler-Code reinschauen. Eine `inline`-Funktion kann natürlich nicht `extern` definiert werden, weil eventuell gar keine Funktion generiert wird, sondern der Code direkt eingefügt wird. Daher ist eine `inline`-Funktion automatisch `static`.

Darüberhinaus gibt es noch einen weiteren Automatismus. Wenn eine Funktion in einer Klasse nicht nur deklariert, sondern auch definiert wird, dann ist sie automatisch inline (und damit static). Das Beispiel `vektor.h` aus Abschnitt 3.1 ist daher auch in seiner ursprünglichen Form korrekt:

```
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); }
};
```

Für derart kurze Funktionen macht dies durchaus Sinn. Die Funktionen in eine cc-Datei auszulagern war insofern ein etwas akademisches Beispiel.

4.4 Aufgaben

1. Schreiben Sie eine statische Funktion, die das Maximum zweier Werte zurückgibt. Muss die Definition der Funktion in der Header- oder in der cc-Datei stehen, wenn man sie in zwei verschiedenen cc-Dateien benutzen will? Probieren Sie aus, ob ihre Überlegung stimmt.

2. Deklarieren Sie ihre Funktion als `inline` und untersuchen Sie, ob das einen Unterschied macht.

Kapitel 5

Parameterübergabe

5.1 Werte- und Referenz-Parameter

In diesem Abschnitt geht es darum, was es bewirkt, wenn in einer Funktion eine Variable geändert wird. Das heißt wir haben die Situation, dass einer Funktion eine Variable übergeben wird und diese auf einen anderen Wert gesetzt wird. In C++ gibt es die Möglichkeit festzulegen, ob sich dadurch der Wert der Variablen außerhalb der Funktion ändern soll:

Wert kopieren	**Referenz**

`Funktion(double Wert)`
`{`
`  Wert = 0.0;`
`}`
	`Funktion(double & Wert)`
	`{`
	`  Wert = 0.0;`
	`}`

Änderung der Kopie außerhalb wirkungslos	Änderung wirkt sich außerhalb aus
Kopieren notwendig	(fast) keine Laufzeiteinbußen

Im ersten Fall wird der Wert kopiert, was natürlich bedeutet, dass innerhalb der Funktion diese Kopie verändert wird. Im zweiten Fall ist die Variable `Wert` eine Referenz der Variable, die beim Funktionsaufruf übergeben wird. Ein Änderung bewirkt daher eine Änderung außerhalb. In C++ gibt es hier keine Unterscheidung zwischen eingebauten Typen und selbstdefinierten Klassen. Man kann also sowohl eingebaute Typen (z.B. `int` oder `double`) wie auch selbstdefinierte Klassen (wie unsere Vektorklasse) als Kopie oder als Referenz übergeben.

Es muss aber davor gewarnt werden, unreflektiert Referenzen zu verändern. Erwartet der Benutzer einer Funktion nicht, dass sich eine übergebene Variable ändert, so kann es sehr lange dauern, bis er diesen Fehler findet. Die Dokumentation und noch besser der Name der Funktion sollte klarstellen, dass der Parameter modifiziert wird.

5.2 Konstante Argumente

Die Übergabe einer Variable bietet nicht nur die Möglichkeit, dass man sie innerhalb der Funktion nach außen sichtbar verändern kann, es hat auch den Vorteil, dass man sich das Kopieren spart. Bei einer Variable vom Typ `double` ist das keine große Ersparnis. Wenn man aber ein Feld mit einer Million `double`s hat, dann wirkt sich das schon aus. Aber auch bei kleineren Datenstrukturen kann dies einen großen Unterschied in der Laufzeit machen, wenn

diese sehr oft kopiert werden müssen. Der Autor hatte einen konkreten Fall, bei dem ein einziges Zeichen – ein „&“ – ein Programm um ein Drittel schneller gemacht hat.

Wenn man allerdings eine Variable als Referenz übergibt um sich das Kopieren zu sparen, birgt dies die Gefahr in sich, dass man die Variable *aus Versehen* ändert, obwohl man dies garnicht möchte. C++ bietet eine Möglichkeit, wie sich der Programmierer dagegen schützen kann. Man kann nämlich angeben, dass sich eine Variable nicht ändern soll. Dies geschieht, indem man `const` davorschreibt:

```
Funktion(double const& Wert)
{
  Wert = 0.0; // nicht mehr möglich
}
```

Man beachte dabei die Reihenfolge. Sie ist von der Variable ausgehend von rechts nach links zu Lesen. Im vorliegenden Fall haben wir eine Referenz, die konstant ist. Umgekehrt, also `double &const Wert`, hätten wir eine Referenz von einer Konstanten, was etwas anderes ist.

Abgesehen von konstanten Argumenten, gibt es noch die Möglichkeit, anzugeben, dass eine Funktion die Klasse selbst nicht ändert. Das ist sinnvoll, da man sonst keine Funktionen von konstanten Variablen aufrufen könnte:

vektor.cc
```
#include <cmath>
#include "vektor.h"
double Vektor::norm() const
  {
    return sqrt(myX*myX+myY*myY);
  }
```

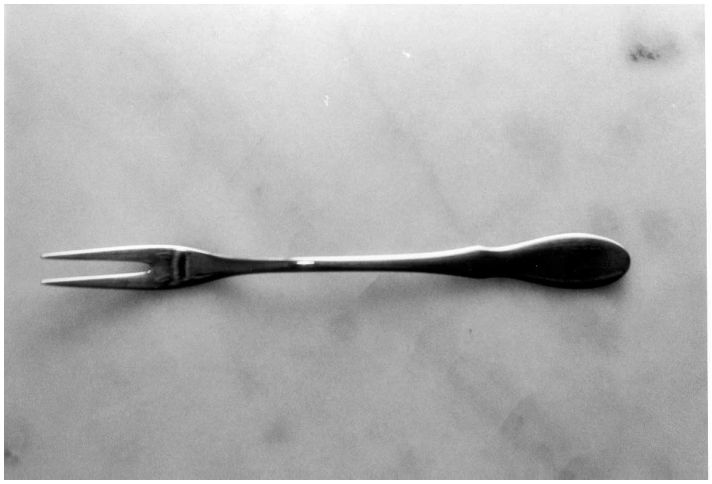
Die Norm eines Vektors kann somit berechnet werden, auch wenn der Vektor eine Konstante ist.

5.3 Kopierkonstruktor

Bei selbstdefinierten Klassen mag es in einigen Fällen nicht sinnvoll sein, den Inhalt der Klasse einfach zu kopieren. Der Programmierer einer Klasse kann daher dem Kompiler mitteilen, welcher Code zum Kopieren verwendet werden soll. Das geschieht, indem man den so genannten *Kopierkonstruktor* definiert, d.h. einen Konstruktor, der als Argument eine konstante Referenz derselben Klasse erhält:

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

  Vektor(Vektor const& v)
    :myX(v.myX),myY(v.myY) {};

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

Wenn kein Kopierkonstruktur angegeben wird, generiert der Kompiler standardmäßig einen Kopierkonstruktor, der alle Elemente der Klasse kopiert.



Man kann (sollte aber nicht) beliebigen Blödsinn damit treiben:

```
Vektor::Vektor(Vektor const& v)
  :myX(v.myX+1),myY(v.myY) {};
```

In diesem Fall würde bei jeder Kopie, die x-Koordinate um eins erhöht werden. Das widerspricht vollkommen der Intuition, die man bei einem Kopierkonstruktor hat. Wir nehmen aber an, dass Sie "erwachsen" genug sind, so etwas nicht zu tun...

5.4 Kopieren und Referenzen bei Initialisierungen

Interessanterweise (oder konsequenterweise) ist es auch möglich, Referenzen außerhalb des Funktionskopfes zu deklarieren, genauso wie man Variablen deklariert. Die Referenz steht dann als Platzhalter für das Original:

Wert kopieren	**Referenz**

{
⋮
`double Kontostand;`
`Kontostand = 1000.0;`
`double Einkommen(Kontostand);`
`Einkommen = 0.0;`
⋮
}
	{
	⋮
	`double Kontostand;`
	`Kontostand = 1000.0;`
	`double & Einkommen(Kontostand);`
	`Einkommen = 0.0;`
	⋮
	}

Kontostand ist 1000	Kontostand ist 0
Änderung der Kopie	Änderung des Originals (und der Referenz)

Einer Referenz kann man nur bei der Initialisierung die Variable zuweisen, die sie referenzieren soll. Wenn man ihr später eine Variable „zuweist“, wird deren Wert in die Variable geschrieben, die referenziert wird.

```
int main ()
{
  double a(100);
  double &b(a);
  double c(200);
  b = c; // a und b sind 200
  b = 500; // a und b sind 500
}
```

Wenn man daher eine Referenz als Mitglied einer Klasse hat, muss diese direkt im Konstruktor initialisiert werden, wie es in Abschnitt 3.1 vorgestellt wurde.

5.5 Aufgaben

1. Ändern Sie ihre Klasse für komplexe Zahlen derart ab, dass überall wo möglich `const&` bzw. `const` verwendet werden. Probieren Sie aus, was passiert, wenn Sie versuchen eine nicht-konstante Funktion auf eine konstante Variable anzuwenden.

2. Haben Sie an die Ausgabefunktion gedacht?

3. Schreiben Sie einen Kopierkonstruktor für ihre Klasse (auch wenn er dem Standardkopierkonstruktor entspricht).

4. Erklären Sie, warum der Kopierkonstruktor sinnvollerweise ein Argument mit `const&` und nicht `const` oder `&` erwartet.

Kapitel 6

Speicherverwaltung

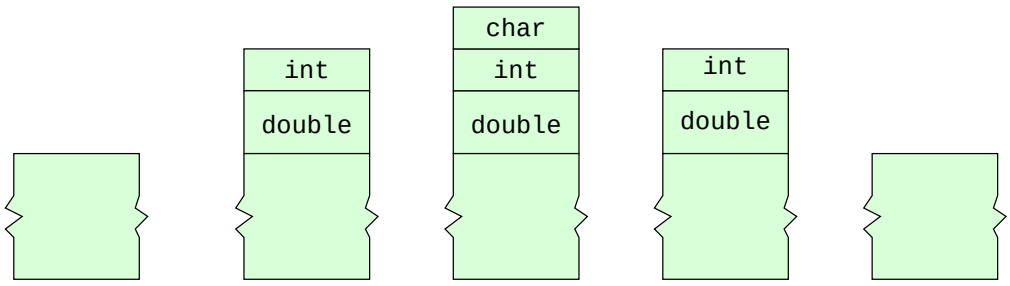
Wie bereits mehrfach angekündigt, hat man mit C++ eine größere Kontrolle über die Speicherverwaltung – aber damit auch eine größere Verantwortung. Man unterscheidet in C++ zwei Arten von Speicher: den Stapel (Stack) und den Haufen (Heap). Während ersterer einfach zu handhaben ist, schließlich ist ein Stapel ein geordnetes Ding, ist es schon schwieriger einen Haufen von Speicherbereichen zu verwalten. Im Zuge dessen werden wir auch mit den vielgefürchteten Zeigern (Pointern) in Berührung kommen. Es hat sich aber gezeigt, dass, wenn man diese ausschließlich dann einsetzt, wenn es nötig ist, und die Datentypen schön kapselt, man auch mit Zeigern gut auskommen kann, ohne sich ständig an ihren Spitzen zu stechen.

6.1 Stapel

Bevor wir uns in die Tiefen der Speicherverwaltung wagen, wollen wir uns dem angenehmeren Teil widmen, mit dem man es zum Glück in den meisten Fällen zu tun hat: dem Stapel. Sobald eine Variable definiert wird, wird vom System dafür Speicher angefordert:

```
void Person::erhoeheEinkommen(double Wert, int )
{
  char c;
      ⋮
}
```

Man kann sich das so vorstellen, dass der angeforderte Speicher wie ein Stapel verwaltet wird. Dabei wird er in umgekehrter Reihenfolge wieder zurückgegeben, wie er angefordert wurde.

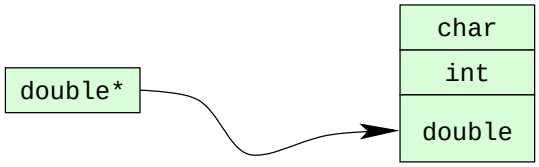


Genauer gesagt wird der Speicher wieder freigegeben, sobald der Gültigkeitsbereich (Scope) der Variable verlassen wird. Im Unterschied zu Java wird der Speicher für die Variable *sofort* freigegeben und nicht zu einem undefinierten Zeitpunkt später. Insbesondere ist es daher nicht nötig, dass eine Garbagge-Collection ausgeführt wird.

Falls vorhanden, wird automatisch vor der Freigabe des Speichers der Destruktor ausgeführt. Die Rolle des Destruktors rückt daher eher in den Mittelpunkt, da genau bekannt ist, wann er ausgeführt wird.

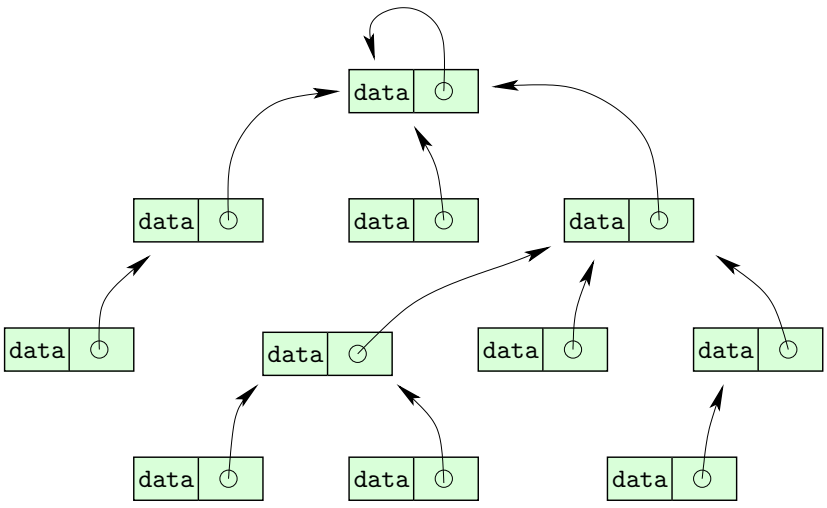
6.2 Zeiger

Die Speicherverwaltung im vorherigen Abschnitt ist einfach und gut, aber was machen wir, wenn wir einen Speicherbereich länger brauchen? Es wäre nicht möglich, eine Funktion zu schreiben, die Speicher anfordert um darin das Ergebnis zurückzugeben. Um dies zu realisieren, benötigen wir aber erst eine Konstruktion, die uns erlaubt auf den Speicherbereich zu *zeigen*, den wir zurückgeben. Die Idee hinter Zeigern ist, dass wir eine Variable haben, die auf eine andere „zeigt“:



Typische Anwendungen von Zeigern sind dynamische Datenstrukturen, wo Zeiger zwischen Teilen der Datenstruktur hin- und herzeigen. Sie sind immer dann sinnvoll, wenn man die Struktur lokal ändern möchte ohne sie komplett neu aufbauen zu müssen. Prominente Beispiele sind:

- verkettete Listen
- dynamische Arrays
- Bäume:



Die Verwendung von Zeigern erläutert man am besten wieder an einem Beispiel:

```
int a = 5;
int b = 7;
int *p;
p = &a;
int *q;
q = &5;// Fehler
int *r;
r = &b;
*p = 9;// a ist nun 9
p = r;// p zeigt nun auf b
*p = 8;// b ist nun 8
```

Bei der Deklaration eines Zeigers setzt man einen Stern vor den Variablennamen, wie dies bei `p` geschieht. Um nun einem Zeiger einen Wert zuweisen zu können brauchen wir noch die Möglichkeit, die Adresse einer Variable zu bekommen. Dies geschieht mit einem vorgestellten `&`, wie man bei der Zuweisung von p sehen kann. Die Zuweisung für q ist deshalb fehlerhaft, da 5 eine Konstante ist.

Letztendlich benötigen wir noch die Möglichkeit, den Inhalt der Variable zu bekommen, auf die ein Zeiger zeigt. Unglücklicherweise geschieht dies wieder mit einem Stern, so dass dieser je nach Kontext eine andere Bedeutung hat. Diese Verwendung sieht man in der drittletzten und in der letzten Zeile. Man sollte sich unbedingt den Unterschied zur vorletzten Zeile klar machen, wo der Zeiger selbst einen neuen Wert bekommt, also auf etwas anderes zeigt.

Zusammenfassend haben wir folgende Symbole benutzt:

`*`	Deklaration von “Pointer auf”
`(&`	Deklaration einer Referenz)
`*`	Inhalt auf den der Pointer zeigt
`&`	Adresse von einer Variablen

Ein Stück zu recht sind Zeiger ein gefürchtetes Werkzeug. Die am schwierigsten zu findenden Fehler haben meist mit Zeigern zu tun. Wenn man einen Zeiger benutzt, der auf einen

ungültigen Bereich zeigt, kann man sich schon mal den Tag versauen. Nichts desto trotz können Zeiger sehr hilfreich sein, wenn man sie gekonnt einsetzt.

Wenn man nun das Verhalten von Zeigern mit dem von Instanzen von Klassen in Java vergleicht, so kann man bis zu einem gewissen Grad sagen, dass in Java alle Variablen Zeiger sind.

6.3 Speicher anfordern: `new`

Nun soll die Möglichkeit vorgestellt werden, wie man Speicher vom System anfordern kann, der nicht automatisch zurückgegeben wird. Der Befehl dazu ist ähnlich wie in Java `new`:

Format
```
type* new type;
```

Damit ist gemeint, dass die Funktion `new` bei einem Parameter `type` einen Zeiger auf diesen Typ zurückgibt. Dabei reserviert es den Speicherbereich, auf den der Zeiger zeigt (wenn noch so viel Speicher verfügbar ist).

Die Benutzung erläutern wir wieder an mehreren Beispielen:

Benutzung
```
#include <string>
int main()
{
  int* MeinZeiger = new int;
  std::string * ZweiterZeiger = new std::string;
  short int * ZeigerMitInitialisierung = new short int(42);
  double const* ZeigerAufKonstante =
    new double const(3.1415926535897);
  return 0;
}
```

Im ersten Fall wird Speicher für ein `int` geholt und der Zeiger `MeinZeiger` damit initialisiert. Ähnlich im zweiten Fall für eine Klasse vom Typ `string`. Die dritte Variable erhält einen Zeiger auf einen Speicherbereich von Type `short int`, der bereits mit 42 initialisiert wurde. Im letzten Fall wird die Technik mit `const` derart kombiniert, dass wir einen Zeiger auf eine Konstante haben. (Wir erinnern uns: Von der Variable ausgehend nach links: „*“ für Zeiger und `const` für Konstante.) Den Wert `*ZeigerAufKonstante` also 3.1415926535897 kann daher nicht verändern. Den `ZeigerAufKonstante` jedoch schon!

6.4 Speicher zurückgeben: `delete`

Wenn wir Speicher mit `new` anfordern, dann sind wir selbst dafür verantwortlich, diesen zurückzugeben. Dies geschieht mit `delete`:

Format:
```
delete PointerToType;
```

Auch hier ein kleines Beispiel, das hoffentlich weitgehend selbsterklärend ist:

```
int main()
{
  int* MeinZeiger = new int;
  double * ZweiterZeiger = new double;
      ⋮
  delete MeinZeiger;
  double * AuchZweiterZeiger = ZweiterZeiger;
  *AuchZweiterZeiger = 1.41;
  delete AuchZweiterZeiger;
  *ZweiterZeiger = 1.73;// Fehler zur Laufzeit!!!
  return 0;
}
```

Die letzte Zeile zeigt eine der Gefahren, die Zeiger so eklig machen: Der Speicher, auf den `ZweiterZeiger` zeigt, ist bereits wieder freigegeben worden. Allerdings existiert der Zeiger noch. Wenn wir nun versuchen darauf zuzugreifen und z.B. wie im gegebenen Fall in den Speicherbereich, auf den der Zeiger zeigt, zu schreiben, ist dies falsch. Das Problem ist, dass dieser Fehler erst zur Laufzeit auftritt und es daher sein kann, dass er erst spät entdeckt wird, z.B. wenn das Programm schon an den Kunden ausgeliefert ist. Was die Sache noch gemeiner macht ist, dass nicht klar ist, wo der Wert nun hingeschrieben wird. Es kann ja sein, dass der Speicherbereich inzwischen anderweitig vergeben wurde. Dann ändert sich urplötzlich eine andere Variable im Programm. Es kann sogar passieren, dass der Speicherblock garnicht mehr diesem Programm zugewiesen wurde. Wenn das Betriebssystem das merkt, wird das Programm meist gnadenlos abgebrochen (Schutzverletzung oder Segmentation fault). Beim Finden solcher Fehler sind Programme sehr hilfreich, die die Benutzung des Heaps mitprotokoliieren. Beispiele dafür sind purify von IBM, valgrind (GNU/Linux) oder die Debug-Bibliothek von Microsoft.

6.5 Felder und Zeiger

Leider sind Zeiger und Felder in C/C++ nah verwandt, was leicht für Verwirrung sorgt. Das geht soweit, dass Felder und Zeiger in vielen Situationen austauschbar sind. Man kann sich unter einem Feld in etwa einen Zeiger auf das erste Element vorstellen.

Das hat zur Folge, dass Felder einen Sonderfall von `delete` sind: Wenn man Speicher von Feldern zurückgibt, muss man nicht `delete`, sondern `delete[]` verwenden.

`Beispiel:`

```
int main()
{
  int* MeinArray = new int[1000];
      ⋮
  delete[] MeinArray;

  return 0;
}
```

Da Zeiger und durch ihre Verwandtschaft damit auch Felder gefährlich sind, sollte man besser die gekapselte Version von Feldern verwenden, wenn es vertretbar ist. Diese Klasse heißt `std::vector` und wird im Abschnitt 9.4 vorgestellt werden.

6.6 Die dunkle Seite der Macht

Da man es nicht oft genug sagen kann, hier noch einmal eine Warnung vor Zeigern. Verwenden Sie Zeiger nur, wenn es nicht anders (sinnvoll) geht, z.B.

- wenn ansonsten massives Kopieren von Speicher notwendig ist,
- wenn man keine Referenz verwenden kann,
- wenn man keine Standard-Container verwenden kann.

Einen Integer oder Double kann man prima kopieren. Auch bei einer handvoll Zahlen lohnt es sich im Allgemeinen nicht, deswegen Zeiger zu bemühen.

Wenn es nur darum geht, eine Abkürzung für einen langen Ausdruck zu haben oder einer Funktion eine große Datenstruktur zu übergeben, dann tut man besser besser daran, eine Referenz zu benutzen. Dann kann es auf jeden Fall nicht passieren, dass man aus Versehen den Zeiger ändert an Stelle des Wertes der Variable, auf die der Zeiger zeigt.

Für Dinge wie Listen, dynamische Arrays oder Bäume gibt es in C++ vorgefertigte Klassen, die man in vielen Fällen prima zu seinen eigenen Datenstrukturen zusammenstecken kann.

Wenn man doch einmal einen Zeiger verwenden muss, dann sollte man ihn

- in einer Klasse einkapseln und
- diese Klasse austesten, bevor man sie verwendet.

6.7 Aufgaben

1. Schreiben Sie ein Programm, das
   - Speicher mit `new` anfordert
   - dem Speicher etwas zuweist
   - dem Zeiger etwas zuweist
   - allen Speicher wieder zurückgibt
2. Erweitern Sie das Programm, so dass mindestens drei Kombinationen von „Zeiger“, „const“ und „Referenz“ verwendet werden. Weisen Sie dabei der Variablen jeweils etwas zu. Generieren Sie für jede ihrer Variablen mindestens einen Fehler.
3. Schreiben Sie eine Klasse, die die Verwendung eines Zeigers kapselt. Ein Beispiel wäre ein Array, das eine Funktion „GrößeAndern“ hat. Sorgen Sie dafür, dass der Destruktor den Speicher wieder frei gibt, wenn er am Ende des Geltungsbereichs automatisch aufgerufen wird.

Kapitel 7

Vererbungslehre und Verwandtes

7.1 Syntax

Ein Großteil des Vererbens funktioniert in C++ genauso wie in Java. Es gibt `private`, `protected` und `public` Funktionen und Variablen. Allerdings ist die Syntax leicht verschieden:

```
class Kind:public Mutter {
  ⋮
}
```

Statt `extends` muss `:public` stehen, wenn man das gleiche Verhalten wie in Java haben möchte: Variablen und Funktionen, die `public` sind, sind überall verwendbar. Die abgeleitete Klasse kann Variablen und Funktionen verwenden, die `protected` sind. Außerhalb der Basisklasse sind Variablen und Funktionen, die `private` sind, nicht zu benützen.

7.2 Intermezzo: Namensräume

Namensräume haben zwar nichts mit Vererbung zu tun, aber sie kapseln ähnlich wie Klassen Namen anderer Objekte (wie Funktionen, Variablen oder Klassen). Sie entsprechen den „Packages“ aus Java und wurden in C++ eingeführt, um dabei zu helfen, wenn man verschiedene Bibliotheken in einem Projekt verwenden will. Dabei tritt es regelmäßig auf, dass ein Name wie `list`, `set`, `string`, `point` oder `vector` in mehr als einer Bibliotheken vergeben wurde. Natürlich passen dabei die definierten Objekte in den seltensten Fällen zusammen.

Ein Ausweg ist die Konvention, jedes Objekt einer Bibliothek `FOO` mit dem Namen der Bibliothek zu beginnen, d.h. man verwendet `FOOlist`, `FOOset`, `FOOpoint` bzw. `FOOvector`. Der offensichliche Nachteil ist, dass es nicht sonderlich schön aussieht und viel zu tippen ist. Namesräume entschärfen das Problem derart, dass man den Präfix nur außerhalb der Bibliothek angeben muss:

```
// Bibliothek
namespace FOO {
  void f() {
    // something useful
  }
}

// Nutzer
int main() {
  FOO::f();
}
```

Auf Elemente des Namensraumes `FOO` greift man also genauso zu, wie auf Bezeichner, die aus einer Klasse stammen, indem man `FOO::` voranstellt. Die Namen aus der Standandbibliothek, die mit jedem C++-Kompiler mitkommt, sind ebenfalls in einem eigenen Namensraum, nämlich `std`. Dies erklärt die etwas kryptischen Bezeichner `std::cout` und `std::endl`.

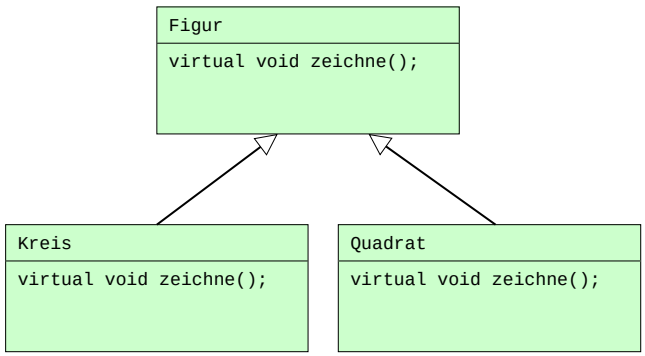
Wenn man ein Objekt (wie Funktion, Variable oder Klasse) sehr oft verwendet, ist jedoch auch das Voranstellen von `FOO` bei der Verwendung lästig. Daher gibt es die Möglichkeit ein für alle mal (in dieser Datei) zu sagen, dass man mit `f` eigentlich `FOO::f` meint:

```
using FOO::f;

int main() {
  f();
}
```

Vor der Verwendung von `using` in Headerdateien muss aber gewarnt werden, da man damit allen Benutzern dieser Headerdatei seine Meinung aufzwingt, dass mit `f` in Wahrheit `FOO:f` gemeint ist.

7.3 Dynamisches und statisches Binden

Zurück zum Vererben. Betrachten wir folgende Situation:



Die Klassen `Kreis` und `Quadrat` sind beide von `Figur` abgeleitet und implementieren die Funktion `zeichne()`. Wenn man nun eine Instanz z.B. von `Kreis` hat, dann ist dies insbesondere eine Figur. Man kann daher eine Referenz von Figur oder einen Zeiger auf Figur damit initialisieren.

```
Kreis MeinKreis;
Figur *Zeiger;
Zeiger = &MeinKreis;
(*Zeiger).zeichne();
```

In der letzten Zeile wird dann die Funktion `zeichne` von `Kreis` aufgerufen. In Java ist dies das Standardverhalten bei Vererbung. In C++ muss man jedoch vor die Funktion in der Basisklasse `virtual` schreiben, damit dieser Mechanismus funktioniert. Damit werden alle Funktionen mit denselben Namen und Parametern in abgeleiteten Klassen automatisch virtuell. Im vorliegenden Beispiel also `zeichne()` in `Kreis` und `Quadrat`. Es schadet aber auch nicht dort ebenfalls `virtual` zu verwenden, sondern erinnert noch einmal daran.

Falls man kein `virtual` verwendet, wird nicht überprüft, von welchem Typ die Klasse in Wirklichkeit ist. Im obigen Beispiel würde daher die Funktion `zeichne` der Basisklasse aufgerufen werden. Man kann (und muss) also zwischen virtuellen und nicht virtuellen Funktionen unterscheiden. Der Vorteil von nicht virtuellen ist derjenige, dass zur Laufzeit nicht überprüft werden muss, welche Klasse vorliegt – Kreis oder Figur. Das spart Rechenzeit. Außerdem muss keine Tabelle mit den Funktionen vorgehalten werden. Das spart Speicherplatz. Wenn man also obigen Mechanismus nicht benötigt, dann ist es von Vorteil, wenn man diese Maschinerie auch nicht anschmeißt.

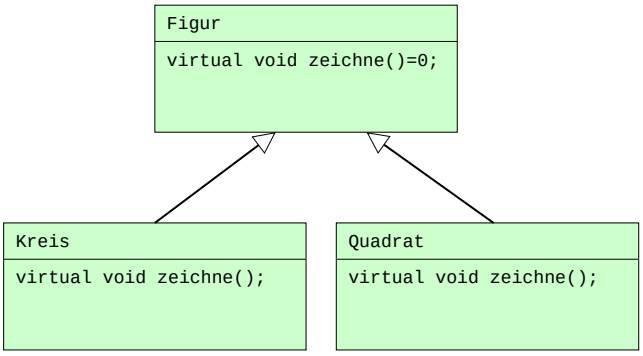
Ein Fall, in dem man ihn quasi immer braucht, ist der Destruktor. Falls der Destruktor nicht virtuell ist, und er für eine abgeleitete Klasse aufgerufen wird, die sich aber hinter einer Variable vom Typ einer Basisklasse verbirgt, dann wird der falsche Destruktor aufgerufen. Merke daher

Destruktoren müssen immer virtuell sein.

Eine Ausnahme von der Regel ist natürlich, wenn von einer Klasse nicht abgeleitet werden soll. (Dies ist z.B. für die Standardklassen `std::vector` und `std::list` der Fall.)

7.4 Abstrakte Klassen

Bei virtuellen Funktionen gibt es noch eine Besonderheit für den Fall, dass man sie eigentlich nicht definieren möchte (wie im obigen Fall `zeichne` für `Figur`). Man kann dies dadurch zum Ausdruck bringen, dass man sie gleich 0 setzen, was gleichbedeutend damit ist, dass sie nicht existieren:



Klassen mit solchen Funktionen nennt man *abstrakte Klassen*. Sie können als Ersatz für Interfaces angesehen werden (die es in C++ nicht gibt).

- Eine Klasse ist automatische eine abstrakte Klasse, sobald es eine virtuelle Funktion gibt, die 0 ist.
- Abstrakte Klassen können nicht instanziiert werden.
- Im Unterschied zu Interfaces können aber Methoden definiert werden. Man kann also Teile der Klasse bereits vorfertigen und andere frei lassen.
- Man kann von einer abstrakten Klasse wieder Klassen ableiten, die ebenfalls abstrakt sind.
- Eine abgeleitete Klasse ist dann nicht mehr abstrakt, wenn alle 0-Funktionen definiert sind.

7.5 Aufgaben

1. Schreiben Sie eine Klasse `Person`, die einen Namen vom Typ `std::string` hat. (Dazu muss man die Header-Datei `string` einbinden.) Leiten Sie davon eine Klasse `Angestellter` ab, der auch noch ein Gehalt vom Type `unsigned int` hat.
2. Schreiben Sie eine virtuelle Funktion, die den Namen bzw. den Namen und das Gehalt ausgibt.
3. Verwenden Sie bei der Deklaration der Klasse `std::string` und bei der Definition mittels `using` nur `string`.

Kapitel 8

Operatoren

Was Mathematiker schon vor Jahrhunderten erfunden haben, gibt es jetzt endlich in ihrer Programmiersprache: Operatoren definieren. Es ist in C++ möglich, Operatoren wie $+$, $-$ oder $*$ für eigene Klassen zu definieren. Unser Beispiel eines Vektors krankt bei der Verwendung der Addition:

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

  Vektor addiere(Vektor v);
};

      ⋮
    VektorA = VektorB.addiere(VektorC);
      ⋮
```

Eigentlich will man (zumindest als Mathematiker) stattdessen lieber schreiben:[1]

```
      ⋮
    VektorA = VektorB + VektorC;
      ⋮
```

Wir werden in Kürze erfahren, wie das geht. Neben der Addition kennt man in C++ (unter anderem) die Verknüpfungen in Tabelle 8.1.

Sehr vorsichtig sollte man mit der Definition von „`()`" sein. Von „`,`" und „`->`" lässt man am besten ganz die Finger (wenn man sie sich nicht verbrennen möchte).

[1] Es existiert mindestens ein Mensch, der das nicht möchte.

+	Typ × Typ	→	Typ	++	Typ	→	Typ
-	Typ × Typ	→	Typ	--	Typ	→	Typ
*	Typ × Typ	→	Typ	+=	Typ × Typ	→	Typ
/	Typ × Typ	→	Typ	-=	Typ × Typ	→	Typ
==	Typ × Typ	→	bool	*=	Typ × Typ	→	Typ
!=	Typ × Typ	→	bool	/=	Typ × Typ	→	Typ
>	Typ × Typ	→	bool	=	Typ × Typ	→	Typ
-	Typ	→	bool	[]	TypA × TypB	→	TypC

Tabelle 8.1: Beispiele für Operatoren in C++

8.1 Syntax für binäre Operatoren

Die Syntax für Operatoren soll am Beispiel eines binären Operators erläutert werden, also eines Operators mit zwei Argumenten. Wir wählen die Addition:

Format:
```
Typ& operator+(Typ const& ZweitesArgument) const;
```

Für unseren Vektor sieht die Deklaration folgendermaßen aus:

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

  Vektor operator+(Vektor const& v) const;
};
```

Die Frage ist nur, wo das erste Argument steckt. Die Funktion `operator+` gehört zu einer Klasse. Für diese wird sie aufgerufen. Dabei ist die Instanz, für die `operator+` aufgerufen wird, das erste Argument, wohingegen das zweite übergeben wird. Das wird vielleicht etwas klarer, wenn man sich die Definition von `Vektor::operator+` ansieht:

Definition:
```
      ⋮
Vektor Vektor::operator+(Vektor const& v) const
{
  return Vektor(myX+v.myX, myY+v.myY);
}
      ⋮
```

Unäre Operatoren werden dann naheliegenderweise entsprechend ohne den Parameter deklariert und definiert.

8.2 Die goldene Regel der drei

Brauchst du eine, brauchst du alle!

Gemeint sind die folgenden Funktionen:

1. Kopierkonstruktor:
   `Vektor::Vektor(Vektor const& v)`
2. Destruktor:
   `Vektor::~Vektor()`
3. Zuweisungsoperator:
   `Vektor& Vektor::operator=(Vektor const& v)`

Es hat sich gezeigt, dass man (fast?) immer all diese drei Funktionen benötigt, wenn man auch nur eine davon implementieren muss. Falls man – aus welchem Grund auch immer – eine der drei Funktionen nicht definieren möchte, sollte man sie `private` deklarieren, aber nicht definieren (und die Deklaration kommentieren). Dadurch erhält man eine Fehlermeldung, wenn man sie doch „aus Versehen“ benutzt.

8.3 Beispiel: Dynamischer Vektor

Um die viele Theorie mit etwas Leben zu füllen, wird in diesem Abschnitt ein Beispiel etwas ausführlicher vorgestellt. Es handelt sich um einen Vektor, bei dem die Dimension bei der Initialisierung festgelegt wird.

Deklaration

```
class Vektor {
private:
  double *data;
  unsigned int groesse;
public:
  Vektor(unsigned int Groesse);
  Vektor(Vektor const& v);
  ~Vektor();
  Vektor& operator=(Vektor const& v);
    ⋮
};
```

Die Definition des Konstruktors sieht dabei folgendermaßen aus: Wir speichern die aktuelle Größe und legen ein Feld dieser Größe an.

```
Vektor::Vektor(unsigned int Groesse)
  {
    groesse = Groesse;
    data = new double[Groesse];
  }
```

Im Fall des Kopierkonstruktors muss man die Größe aus dem alten Vektor holen. Außerdem kopiert man den Inhalt des Feldes:

```
Vektor::Vektor(Vektor const& v)
  {
    groesse = v.groesse;
    data = new double[groesse];
    for (unsigned int position=0;
        position<groesse; ++position)
      { data[position] = v.data[position]; }
  }
```

Im Destruktor wird der Speicherbereich wieder freigegeben, den man belegt hat:

```
Vektor::~Vektor()
  {
    delete[] data;
  }
```

Beim Zuweisungsoperator muss man zuerst den Speicher freigeben, den der alte Vektor belegt hat. Anschließend reserviert man den Speicher für die Kopie und führt das Kopieren durch:

```
Vektor& Vektor::operator=(Vektor const& v)
  {
    delete[] data;
    groesse = v.groesse;
    data = new double[groesse];
    for (unsigned int position=0;
        position<groesse; ++position)
      { data[position] = v.data[position]; }
    return *this;
  }
```

So gesehen ist eine Zuweisung die Kombination von Destruktor und Kopierkonstruktor. In anderen Fällen ist es aber möglich, die Zuweisung effizienter zu machen – darum gibt es die Möglichkeit, sie selbst zu definieren. Wir geben beim Zuweisungoperator eine Referenz von dem Objekt zurück, das wir verändert haben. Dadurch sind Zuweisungen wie

```
  a = b = c;
```

möglich. Diese Zuweisungskette wird von rechts nach links abgearbeitet. (Warum wohl?) Damit ist das Ergebnis von `b = c` der neue Inhalt von `b`. Dieser kann dann `a` zugewiesen werden. Es ist sinnvoll, eine Referenz zurückzugeben, damit man sich eine (unnötige) Kopie spart. Beachten Sie, dass in C++ die Variable `this` ein Zeiger auf das aktuelle Objekt ist. Man muss daher `*this` zurückgeben.

Die Addition soll uns als Beispiel für einen binären Operator herhalten. Wir lassen nur die Addition von gleichlangen Vektoren zu.

```
Vektor Vektor::operator+(Vektor const& v) const
  {
    if (groesse!=v.groesse)
      { throw "falsche Laenge"; }

    Vektor Ergebnis(groesse);
    for (unsigned int position=0;
        position<groesse; ++position)
      {
      Ergebnis.data[position] =
        data[position]+v.data[position];
      }
    return Ergebnis;
  }
```

Beachten Sie, dass bei der Rückgabe der Gültigkeitsbereich der Variable `Ergebnis` verlassen wird. Die Variable existiert danach nicht mehr! Bei der Rückgabe wird daher mit dem Kopierkonstruktor eine Kopie erstellt. Diese Kopie ist zwar eigentlich unnötig und daher unerwünscht, ohne ausgefuchste Konstruktionen ist dies aber der Preis, den man für die schöne Syntax zahlen muss.

Zu guter Letzt untersuchen wir ein Beispiel dafür, dass zwischen konstanten und nichtkonstanten Funktionen unterschieden wird. Der Operator „`[]`“ wird in beiden Varianten definiert. Die erste ist im Gegensatz zur zweiten nicht konstant.

```
double&
Vektor::operator[](unsigned int const index)
  { return data[index]; }

double const&
Vektor::operator[](unsigned int const index) const
  { return data[index]; }
```

Die Verwendung des Operators „`[]`“ ist diejenige, die man von Feldern kennt. Man kann also mit `A[5]` das fünfte Feld unseres Vektors ansprechen. Falls der Vektor dabei keine Konstante ist, wird die erste Variante verwendet. Die Funktion gibt eine Referenz des entsprechenden Feldes zurück. Man kann daher den Inhalt des Feldes ändern: `A[5] = 17`.

Falls jedoch der Vektor konstant ist, so wird die zweite Variante benutzt. Man erhält eine Konstante, die man nicht ändern kann. So etwas wie `A[5] = 17` ist dann nicht möglich, was ja auch sinnvoll ist, wenn der Vektor konstant ist.

8.4 Warnung

Operatoren sollten nur dann definiert werden, wenn ihre Semantik klar ist!

Selbst wenn es einem im Augenblick praktisch erscheint, sich irgendwelche Operatoren zu definieren, so kann man damit später böse Überraschungen erleben.

Schlechtes Beispiel:

```
bool
Komplex::operator<(Komplex const& c) const
  {
    return realteil<c.realteil;
  }
```

Da sich die komplexen Zahlen nicht anordnen lassen, ist es nicht sehr intuitiv dafür das Kleinerzeichen zu definieren.

8.5 Binäreren Operator außerhalb der Klasse

Abgesehen von der obigen Möglichkeit gibt es noch eine zweite Methode, binäre Operatoren neu zu definieren. Der Operator wird nicht als Funktion der Klasse definiert, sondern als „globale“ Funktion mit zwei Argumenten:

```
Vektor operator-(Vektor const& v1, Vektor const& v2)
  {
    if (v1.Groesse()!=v2.Groesse())
      { throw "falsche Laenge"; }

    Vektor Ergebnis(v1.Groesse());
    for (unsigned int position=0;
        position<v1.Groesse(); ++position)
      {
        Ergebnis[position] = v1[position]-v2[position];
      }
    return Ergebnis;
  }
```

Da nun die Funktion nicht mehr zur Klasse gehört, müssen alle Variablen und Funktionen, die benützt werden, `public` sein.[2]

8.6 Ein- und Ausgabe

Eine Anwendung der alternativen Definition eines binäreren Operators ist die (Ein- und) Ausgabe. Konkret kann man den Operator `<<` so definieren, dass wir ihn für die Ausgabe verwenden können, wie man es von anderen Typen gewöhnt ist:

```
#include <iostream>
#include "vektor.h"

int main() {
  int drei = 3; // int drei(3);
  double pi = 3.14;
  std::cout << drei << " + " << pi << std::endl;

  Vektor v(7);
  for (int w=0; w<7; ++w) { v[w] = w; }
  std::cout << v << " ist der Inhalt von v " << std::endl;

  return 0;
}
```

Dazu muss man wissen, dass `std::cout` vom Typ `std::ostream` ist. Wenn wir `std::cout << v` schreiben, dann soll der Operator `operator<<(std::ostream&, Vektor const&)` angewendet werden:

[2] Eine andere Möglichkeit ist, die Funktion als `friend` zu deklarieren, was aber kontrovers diskutiert wird.

```
std::ostream&
operator<<(std::ostream& ausgabe, Vektor const& v)
  {
    ausgabe << '(';
    for (unsigned int position=0;
        position<v.Groesse(); ++position)
      {
        ausgabe << v[position] << ' ';
      }
    ausgabe << ')';
    return ausgabe;
  }
```

Das erste Argument `ausgabe` ist in unserem Fall `std::cout`, das zweite unser Vektor `v`. Die Routine schreibt nun den Inhalt des Vektors nach `ausgabe`. Das clevere an der Konstruktion ist nun, dass wir eine Referenz der `ausgabe` zurückgeben. Das Ergebnis der Funktion ist also vom Typ `std::ostream&`. Daher kann man den Operator `<<` auf das Ergebnis anwenden und z.B. `std::cout << v << " ist der Inhalt von v " << std::endl;` schreiben.

Das schöne an unserem Operator ist, dass wir ihn auch gleich verwenden können, um in eine Datei zu schreiben:

```
#include <fstream>
#include "vektor.h"

int main() {
  int drei = 3; // int drei(3);
  double pi = 3.14;

  Vektor v(7);
  for (int w=0; w<7; ++w) { v[w] = w; }

  std::ofstream Ausgabe("MeineDatei.txt");
  Ausgabe << drei << " + " << pi << std::endl;
  Ausgabe << " v ist " << v << std::endl;

  return 0;
}
```

Die Klasse `std::ofstream` kapselt eine Datei, in die geschrieben werden kann. Da sie von `std::ostream` abgeleitet ist, funktioniert unser Ausgabeoperator dafür.

8.7 Aufgaben

1. Ändern Sie in Ihrer Klasse für komplexe Zahlen die Funktionen für Addition, Subtraktion, Multiplikation und Division in Operatoren.

2. Erklären Sie, warum Sie den Operator `operator<<` nicht für eine Klasse definieren können.

3. Schreiben Sie die Ausgaberoutine in einer ihrer Klassen als Operator um. Probieren Sie, die Ausgabe mit anderen Typen zu kombinieren sowie die Ausgabe in eine Datei.

4. Wenden Sie die goldene Regel der Drei auf ihr dynamisches Array an.

Kapitel 9

Generisches Programmieren

9.1 Generische Funktionen

Eine der großen Errungenschaften der Mathematik ist die Fähigkeit, von konkreten Anwendungen zu abstrahieren und Probleme auf allgemeiner Ebene zu behandeln. Ein Satz für Vektorräume gilt für alle Vektorräume, egal ob es sich um $\mathbb{R}^n$, $\mathbb{F}_2^m$ oder Polynome handelt. Man muss daher nicht den selben Satz immer und immer wieder neu Beweisen.

Ebenso schön ist es, wenn man beim Programmieren nicht immer und immer wieder dieselben Funktionen schreiben muss. Betrachten wir beispielsweise die Funktion „Absolutbetrag“:

```
int abs(int i)
{ return i<0 ? -i : i; }

float abs(float r)
{ return r<0 ? -r : r; }

double abs(double d)
{ return d<0 ? -d : d; }
```

Bis auf den verwendeten Typ, sind diese Funktionen strukturell identisch. Man verwendet, dass die Typen `int`, `float` und `double` folgende Gemeinsamkeiten haben:

- `operator<` zum Vergleichen
- unärer `operator-`, damit das Negative zurückgegeben werden kann
- einen Konstruktor, der eine 0 als Parameter akzeptiert
- einen Kopierkonstruktor für die Rückgabe

Wenn ein weiterer Typ ebenfalls diese Voraussetzungen erfüllt, kann man die Funktion wieder komplett abschreiben. Es liegt daher nahe, den Computer diese Arbeit machen zu lassen. Wir schreiben daher nur noch eine Schablone (template) für die Funktion. Der Typ wird zum so genannten Template-Parameter:

```
template<typename T>
T abs(T x)
{ return x<0 ? -x : x; }
```

Anstelle des speziellen Typs ist nun der Platzhalter `T` getreten. Die Funktion kann (und wird) für alle Typen generiert („instanziiert“) werden, für die die nötigen Funktionen (<, -, 0, Kopie) vorhanden sind. Der Kompiler ist natürlich nicht schlau genug zu überprüfen, ob die Instanziierung Sinn macht. Falls der Operator < zum Beispiel für komplexe Zahlen definiert wäre, würde eine Funktion dafür generiert werden, die etwas anderes als den gewohnten Absolutbetrag zurückgibt.

Die gängigen Kompiler können (derzeit) eine Funktion nur dann generieren, wenn sie die Definition kennen. Insbesondere reicht also die Deklaration der Template-Funktion bzw. -Klasse nicht aus. Eine gängige Methode diese Schwäche zu umgehen ist, alle Template-Funktionen und -Klassen `inline` und damit `static` in der Header-Datei zu definieren.[1]

9.2 Generische Klassen

Was für Funktionen recht ist, ist für Klassen billigt. Gerade Containerklassen wie Listen, Bäume oder Hashtabellen schreien danach, dass man sie einmal schreibt und für verschiedenen Inhalte verwendet. Unter Java ist man dafür leider bisher dazu gezwungen, den Typ zu casten. Dann wird aber erst zur Laufzeit (z.B. wenn das Programm bereits an den Kunden ausgeliefert ist) der Fehler erkannt. Dank Templates können wir auf Casts verzichten und erweitern die Klasse `Vektor` aus Kapitel 8.3 für allgemeine Elemente:

Templateklasse - Deklaration

```
template<typename T> class Vektor {
private:
  T *data;
  unsigned int groesse;
public:
  Vektor(unsigned int Groesse);
  Vektor(Vektor<T> const& v);
  ~Vektor();
  Vektor<T>& operator=(Vektor<T> const& v);

  Vektor<T> operator+(Vektor<T> const& v) const ;
  T const& operator[](unsigned int const index) const ;
  T & operator[](unsigned int const index) ;
  unsigned int Groesse() const
  { return groesse; }
};
```

Der Klassenname erhält nun als Anhängsel den Typ, für den er betrachtet wird. Gekennzeichnet wird er durch spitze Klammern. Ohne diese Kennzeichnung hätte der Kompiler keine

[1] In Zukunft muss dann der Linker den Kompiler anstoßen, um den Code für eine verlangte Funktion zu generieren.

Chance zu erraten, welchen Typ man denn meint. Bei der Definition der Klassenfunktionen steht der Typ noch nicht fest. Daher werden sie als Templatefunktionen definiert:

Templateklasse - Definitionen

```
template<typename T>
Vektor<T>::Vektor(unsigned int Groesse)
{
  groesse = Groesse;
  data = new T[Groesse];
}

template<typename T>
Vektor<T> Vektor<T>::operator+(Vektor<T> const& v) const
{
  if (groesse!=v.groesse)
    { throw "falsche Laenge"; }
  Vektor<T> Ergebnis(groesse);
  for (unsigned int position=0;
      position<groesse; ++position)
    {
      Ergebnis[position] = data[position]+v[position];
    }
  return Ergebnis;
}

template<typename T>
T const& Vektor<T>::operator[](unsigned int const index)
const
{ return data[index]; }
template<typename T>
T & Vektor<T>::operator[](unsigned int const index)
{ return data[index]; }
```

Beachten Sie, dass auch die Klasse beim Konstruktor eine Kennzeichnung mit `<T>` hat, der Name des Konstruktors aber nicht. Verwendet werden kann diese Klasse dann wie zu vermuten mit

```
    ⋮
  Vektor<double> MeinVektor(7);
  MeinVektor[4] = 3.14;
    ⋮
```

9.3 Templates – Für und Wider

Es gibt einige Gründe, die für die Verwendung von Templates sprechen:

- Durch die generische Programmierung definiert man *eine* Klasse oder Funktion für *viele* Typen.
- Dabei hat man Typsicherheit, das heißt dass *zur Kompilierzeit* sichergestellt wird, dass die verwendeten Typen die notwendigen Funktionen bereitstellen und der Typ einer Variable der richtige ist. (Es sind keine Casts notwendig.)
- Die Verwendung der Klassen oder Funktionen ist auch auch möglich, wenn eine (sinnvolle) gemeinsame Basisklasse fehlt. Einzig die verwendeten Funktionen und Variablen müssen vorhanden sein.

Es darf aber auch nicht verschwiegen werden, dass die Verwendung von Templates einige Nachteile hat:

- Die Kompilierzeiten steigen deutlich an.
- Die Funktionsnamen werden unhandlich lang.
- Man bekommt kryptische Fehlermeldungen (die selten in eine Zeile passen).
- Das Programm wird aufgebläht (wenn auch nicht der Quelltext).

9.4 Standard Template Library

Mit C++ kommen eine Menge Template-Klassen mit, die sich als sehr praktisch erweisen. Gemeint sind vor allem Containerklassen für Datenstrukturen wie doppelt verkettete Listen (`list`), dynamische Felder (`vector`) und balancierte Suchbäume (`map`). Aus diesen Datenstrukturen lassen sich dann sehr leicht kompliziertere zusammenstecken. So erhält man schnell eine Klasse für Graphen:

```
#include <vector>
#include <list>

using std::vector;
using std::list;

class Graph {
  struct Node {
    typedef Node* NeighbourType;
    typedef list<NeighbourType> NeighbourContainer;
    NeighbourContainer neighbours;
  };

  vector<Node> nodes;
};
```

Natürlich muss man sich noch um Funktionen kümmern, die etwa Knoten oder Kanten einfügen. Die wesentliche Struktur steht aber bereits. So stellen `list` und `vector` beide die

Funktion `push_back` zum hinten Anfügen zur Verfügung. Für `vector` ist der Operator `[]` definiert.

Eine Besonderheit bei diesen Containerklassen ist eine Konstruktion, die *Iterator* genannt wird. Unter einem Iterator kann man sich so etwas ähnliches wie einen Zeiger vorstellen:

- Der Operator `operator*` liefert den Zugriff auf das Element, auf den der Iterator zeigt.
- Mit dem Operator `operator++` kann man zum nächsten Element im Container weiterschalten.

Zur Verwendung von Iteratoren haben die Containerklassen zwei wichtige Funktionen:

`begin()`: Iterator, der auf das erste Element zeigt

`end()`: Iterator, der auf das erste Element *nach* dem Ende zeigt

Die seltsame Definition für `end()` führt dazu, dass man Iteratoren wie folgt verwendet:

```
int Graph::Node::degree() {
  int number=0;
  for (NeighbourContainer::iterator
        It = neighbours.begin();
       It != neighbours.end();
       ++It)
    { ++number; }
  return number;
}
```

`list<Node*>::iterator` ist dabei der Typ des Iterators, der zur Klasse `list<Node*>` gehört. In der Schleife wird ein Iterator initialisiert, so dass er auf das erste Element des Containers zeigt. Dann wird die Schleife solange durchlaufen und der Iterator weitergeschaltet, bis er auf das Element nach dem letzten zeigt. Wir zählen dabei im konkreten Fall mit, wieviele Elemente wir besuchen.[2]

Das Unschöne am letzten Beispiel ist, dass der Knoten nicht verändert wird, wir aber einen nicht-konstanten Knoten benötigen, um den Iterator zu benutzen. Für diesen Fall gibt es eine zweite Art von Iteratoren, der für konstante Container funktioniert. (Das ist etwas anderes als ein konstanter Iterator!) Der Iterator heißt `const_iterator`:

```
int Graph::Node::degree() const {
  int number=0;
  for (NeighbourContainer::const_iterator
        It = neighbours.begin();
       It != neighbours.end();
       ++It)
    { ++number; }
  return number;
}
```

[2]Natürlich wäre es in diesem Fall einfacher und effizienter, die Funktion `neighbours.size()` zu verwenden, deren Laufzeit darüberhinaus konstant ist.

Nun wollen wir aber noch die Zeiger in unserem Graphen loswerden. In Wahrheit benötigt man nämlich nur einen Iterator, der weniger kann und daher weniger gefährlich ist (d.h. man tut nicht aus Versehen etwas damit, was man eigentlich nicht wollte). Mit einem Iterator sind unser Graph nun so aus:

```
class Graph {
  struct Node {
    typedef vector<Node>::iterator NeighbourType;
    typedef list<NeighbourType> NeighbourContainer;
    NeighbourContainer neighbours;

    int degree() const;
  };

  vector<Node> nodes;
};
```

Treiben wir das Ganze noch auf die Spitze. Wir wollen uns nicht festlegen, welchen Container wir für die Nachbarn verwenden, `list`, `vector` oder sonst irgendeinen. Daher übergeben wir der Graphklasse ein *template template Argument*.

```
template<template<typename T> class CONTAINER>
struct Graph {
  struct Node {
    typedef typename vector<Node>::iterator NeighbourType;
    typedef CONTAINER<NeighbourType> NeighbourContainer;
    NeighbourContainer neighbours;

    int degree() const {
      int number=0;
      for (typename NeighbourContainer::const_iterator
            it = neighbours.begin();
           it != neighbours.end();
           ++it)
        { ++number; }
      return number;
    }
  };

  vector<Node> nodes;
};
```

Das Templateargument `CONTAINER` ist selbst eine Templateklasse. Wir können nun z.B. `Graph<list>` oder `Graph<vector>` instanziieren. Der Parameter `T`, der kennzeichnet, dass `CONTAINER` eine Template-Klasse ist, dient als reiner Platzhalter und wird sonst nirgends verwendet.

Bei diesem Beispiel stolpern wir über eine weitere Besonderheit: Wir wollen einen Typ verwenden, der in einer Klasse definiert ist, die von einem Template-Argument abhängt –

nämlich `vector<Node>::iterator` bzw. `Node::NeighbourContainer::const_iterator`. In diesem Fall muss dies dadurch kenntlich gemacht werden, dass davor `typename` steht.

9.5 Weitere Möglichkeiten

Die Anzahl der Template-Parameter ist erwartungsgemäß nicht auf einen beschränkt. Bei *mehreren Template-Parametern* werden diese wie gehabt durch Kommata getrennt.

Ähnlich wie bei Parametern von Funktionen, kann man auch bei Templates einen *Standardparameter* festlegen. Die Syntax hierfür ist dieselbe: Man bestimmt durch ein Gleichheitszeichen gefolgt von einem Typ.

Templateklasse mit Standardparameter

```
template<typename T = double>
class Vektor {
private:
  T *data;
    ⋮
```

Man kann somit `Vektor` ohne `<double>` verwenden und bekommt wieder den alten Vektor, d.h. den Spezialfall für `double`.

Letztlich soll noch erwähnt werden, dass Template-Parameter nicht nur Typen sondern auch *ganzzahlige Werte* sein können (z.B. 1, 2, 3 vom Typ `int`). Man kann damit z.B. Vektoren fester Länge definieren:

Templateklasse mit Integer

```
template<int L, typename T = double>
class Vektor {
private:
  T data[L];
    ⋮
```

Die Variable `Vektor<3,int> v` ist dann ein dreidimensionaler Vektor, der drei `int`s enthält. Der Unterschied zu `int v[3]` ist der, dass der Vektor in einer Klasse gekapselt ist, die

- sich um Kopieren und Zuweisung kümmern kann,
- weitere Funktionen bereitstellen kann und
- nicht aus Versehen mit `Vektor<4,int>` verwurschtelt werden kann.

In der Standard Template Library ist `bitset` sein prominentes Beispiel für eine Menge von Bits fester Größe (die Teilmengen einer Grundmenge modellieren).

9.6 Spezialitäten

Es kann den Fall geben, dass eine Template-Klasse oder -Funktion für spezielle Typen nicht instanziiert werden kann, weil eine Voraussetzung nicht erfüllt ist (i.e. eine notwendige Funktion nicht existiert). Noch unangenehmer ist der Fall, dass die Klasse oder Funktion zwar instanziiert werden kann, sie aber nicht so effizient oder gar falsch ist. In diesen Fällen hat

man die Möglichkeit, den Spezialfall für diese Funktion separat zu schreiben. Der Kompiler nimmt immer die speziellste Variante, die er für die Template-Parameter finden kann. Vielleicht wollen wir ja den Fall eines zweidimensionalen Vektors gesondert behandeln:

Templateklasse Spezialisierung

```
template<typename T = double>
class Vektor<2,L> {
private:
  T x,y;
    ⋮
```

Es gibt dabei sogar die Möglichkeit, Template-Klassen rekursiv zu deklarieren. Durch eine Spezialisierung stellt man sicher, dass die Rekursion terminiert. Ein Beispiel dafür wäre ein mehrdimensionales Array:

Rekursives Array

```
template<int DIM, int LENGTH, class TYPE>
class Array{
private:
  Array<DIM-1, LENGTH, TYPE> data[LENGTH];
public:
  Array<DIM-1, LENGTH, TYPE>& operator[](int pos)
    {return data[pos];} };

template<int LENGTH, class TYPE>
class Array<1, LENGTH, TYPE>{
private:
  TYPE data[LENGTH];
public:
  TYPE& operator[](int pos)
    {return data[pos];}
};
```

Man kann sich aber auch eine Template-Klasse `IF` definieren, die einen Typ bereitstellt, der von einer Bedingung abhängt:

IF - Definition

```
template<bool Cond, typename A, typename B>
struct IF {
  typedef A RET;
};

template<typename A, typename B>
struct IF<false,A,B> {
  typedef B RET;
};
```

Diese Template-Klasse `IF` kann verwendet werden um *zur Kompilierzeit* zu entscheiden, ob in Abhängigkeit von der Bedingung `Cond` der Typ `A` oder der Typ `B` verwendet wird. Der "Rückgabewert" dieser Metafunktion `IF` ist der Typ `RET`. Betrachten Sie beispielsweise die Definition

IF - Verwendung

```
IF< (sizeof(int)<4), long, int>::RET a;
```

Wenn die Größe von `int` kleiner als 4 ist, wird die allgemeine Definition von `IF` benutzt. Daher ist `a` vom Typ `long`. Wenn `int` größer oder gleich 4 ist, wird die Spezialisierung von `IF` benutzt und `a` ist vom Typ `int`.

Durch die Kombination von Rekursion und Spezialisierung kann man sich beliebig komplizierte Konstruktionen ausdenken. Beispiele wären:

- Eine Kontainerklasse, die intern Pointer verwendet, wenn die Elemente eine bestimmte Größe übersteigen.

```
IF< (sizeof(long)>sizeof(long*)), long*, long> b;
```

- Eine Funktion `ZZ<m>` $\times$ `ZZ<n>` $\rightarrow$ `ZZ<m*n>`, die nur dann definiert ist, wenn n und m teilerfremd sind.

- Generische Implementation von Designpattern.

In der Tat ist es so, dass man durch Templates eine eigene Programmiersprache hat, die aber zur Kompilierzeit ausgeführt wird. Genauer gesagt ist der Templatemechanismus *Turing-vollständig*. Da einmal definierte Templates nicht wieder neu definiert werden können, ist diese Programmiersprache nicht imperativ sondern funktional.

9.7 Aufgaben

1. Schreiben Sie ihre Maximumsfunktion als Template-Funktion um. Probieren Sie sie mit 4 verschiedenen Typen aus. Überlegen Sie sich, warum Sie den Template-Parameter dabei nicht angeben müssen.

2. Ändern Sie ihr dynamisches Array derart ab, dass der Typ der Elemente ein Template-Parameter ist. Probieren Sie die Funktionen mit zwei unterschiedlichen Typen aus.

3. Übernehmen Sie die generische Funktion für den Absolutbetrag. Verifizieren Sie, dass sie für ihre Klasse der komplexen Zahlen nicht funktioniert. Es ist in C++ nicht möglich, eine Spezialisierung für Template-Funktionen zu schreiben. Überlegen Sie sich, wie man trotzdem (einfach) erreichen kann, dass für komplexe Zahlen eine andere Funktion aufgerufen wird.

4. Schreiben Sie eine Template-Funktion, die das größte Element in einem Container sucht. Probieren Sie Ihre Funktion für eine Liste und einen Vector aus.

5. Lesen Sie die letzten vier angegebenen Texte aus Abschnitt 1.5.[3]

[3] ;-)