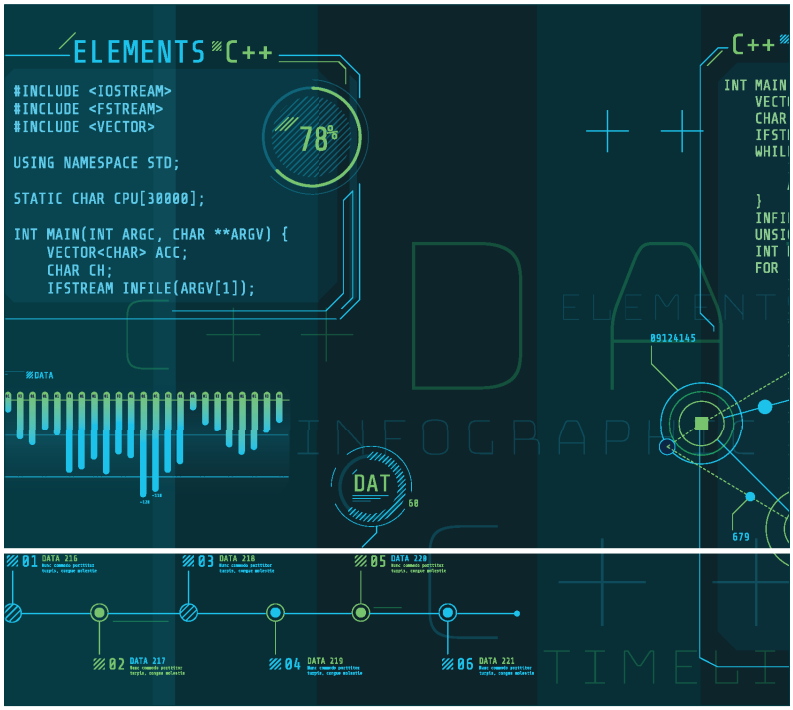
Uwe Probst

Objektorientiertes Programmieren

Eine Einführung für die Ingenieurwissenschaften in C++



2., aktualisierte und erweiterte Auflage

HANSER

Probst

Objektorientiertes Programmieren

Uwe Probst

Objektorientiertes Programmieren

Eine Einführung für die Ingenieurwissenschaften in C++

2., aktualisierte und erweiterte Auflage

HANSER

Dieses Buch ist in der Erstauflage unter dem Titel „Objektorientiertes Programmieren für Ingenieure – Anwendungen und Beispiele in C++“ erschienen.

Der Autor:

Prof. Dr.-Ing. Uwe Probst
Fachbereich Elektro- und Informationstechnik, Technische Hochschule Mittelhessen

Bibliografische Information der Deutschen Nationalbibliothek:
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet unter http://dnb.d-nb.de abrufbar.

Lektorat: Frank Katzenmayer
Herstellung: Frauke Schafft
Coverkonzept: Marc Müller-Bremer, www.rebranding.de, München
Titelmotiv: © shutterstock/ConceptCafe
Satz: Eberl & Koesel Studio, Kempten
Druck und Bindung: CPI books GmbH, Leck
Printed in Germany

Print-ISBN: 978-3-446-47651-6
E-Book-ISBN: 978-3-446-47838-1

Vorwort

In den vergangenen Jahren haben sich Bachelor- und Masterprogramme in vielen Fachrichtungen an den deutschen Hochschulen etabliert. Der Stellenwert studentischer Eigenarbeit hat sich durch die Verkürzung der Präsenzphasen im Zuge des Bologna-Prozesses sowie durch tiefgreifende Umwälzungen im Zuge der Corona-Pandemie stetig gesteigert. Ein solches Selbststudium anhand von übersichtlichen Übungen zielgerichtet zu strukturieren sowie mit klaren Beispielen zu begleiten, ist wesentliches Ziel der zweiten Auflage dieses Buches. Die Inhalte basieren auf den Modulen „Softwareentwicklung“ und „Höhere Informatik“, die ich seit vielen Jahren für Bachelor- und Masterstudierende der Elektrotechnik an der Technischen Hochschule Mittelhessen anbiete.

Die ersten vier Kapitel widmen sich den grundlegenden Paradigmen der objektorientierten Programmierung, erläutern Schlüsselkonzepte anhand des Vergleiches von C++ mit der rein prozeduralen Sprache C und illustrieren wichtige Anwendungsbereiche anhand zahlreicher Beispiele. Ein wichtiges Hilfsmittel der Softwareentwicklung ist die Beschreibungssprache UML, die im Entwicklungsprozess durchgängig verwendet wird. Die bewusste Konzentration auf beispielhafte Aufgabenstellungen aus der Welt der Elektroingenieur:innen soll die praktische Anwendung der vorgestellten Programmiertechniken sowie der UML-Diagramme in diesen Themengebieten veranschaulichen.

Im fünften Kapitel stehen Programmiertechniken im Vordergrund. Zunächst werden die sogenannten fünf Phasen der Softwareentwicklung vorgestellt, in die ein Softwareprojekt grundsätzlich gegliedert werden sollte. In diesem Zusammenhang werden auch Vorgehensmodelle für die Softwareentwicklung besprochen, die zurzeit kommerzielle Anwendung finden. Sie legen fest, in welcher Reihenfolge die einzelnen Phasen durchlaufen werden und in welcher Beziehung sie zueinander stehen.

Eine bewährte Methode professioneller Softwareentwicklung ist es, vorhandene und geprüfte Lösungsansätze auf neue Aufgabenstellungen zu übertragen. Diese Technik ist unter dem Oberbegriff Entwurfsmuster bekannt. Anhand von Problemstellungen wird die Grundidee solcher Muster am Beispiel von drei Vertretern erläutert.

Wichtige Klassenbibliotheken sind Gegenstand des sechsten Kapitels. Zunächst wird das Konzept der Templates (Vorlagen, Schablonen) eingeführt. Sie bilden die Basis der Klassen und Algorithmen in der Standard Template Library (STL). Großen Raum in diesem Abschnitt nimmt die Erstellung grafischer Nutzeroberflächen (Graphical User Interfaces, GUIs) ein. Sogenannte GUI-Klassen sind jedoch nicht standardisiert, sondern stehen in proprietären Bibliotheken unterschiedlicher Hersteller zur Verfügung. In diesem Buch werden die Microsoft Foundation Classes (MFC) verwendet. Den Abschluss bildet ein Abschnitt zum Model-View-Controller-Muster (MVC), das seine Stärke in der Entwicklung von Anwendungen mit grafischer Nutzeroberfläche zeigt.

Im siebten Kapitel werden die Lerninhalte aller vorausgehenden Abschnitte auf mehrere Aufgabenstellungen angewandt. Bei diesen Programmen handelt es sich durchweg um Applikationen, welche die Leser:innen während ihres Studiums einsetzen und ggf. anhand eigener Fragestellungen und Interessen erweitern können.

Für alle behandelten Themengebiete enthält das Buch neben vielen Beispielen eine Fülle von Übungsaufgaben mit Lösungen. Wesentliche Teile der Lösungen sind am Ende des zugehörigen Hauptkapitels abgedruckt, ausführliche Lösungsvorschläge können Sie online abrufen. Zudem finden Sie die Programmierübungen als komplette kompilierfähige Projekte für Microsoft Visual Studio Community 2022 zu diesem Buch auf den unten genannten Websites. Die Kompilierung mit anderen Versionen dieses Entwicklungssystems stellt kein Problem dar, da die Projekte entsprechend konvertiert werden können. Sofern lediglich Compiler anderer Hersteller verfügbar sind, können die Quellcode- und Headerdateien der Konsolenanwendungen aus den Projekten isoliert und in die anderen Systeme übertragen werden. Für grafische Oberflächen geht dies aufgrund der herstellerspezifischen Klassenbibliotheken allerdings nicht.

Dieses Buch richtet sich gleichermaßen an Studierende der Elektrotechnik und Mitarbeiter:innen an Universitäten und Fachhochschulen sowie an Ingenieur:innen in der Praxis, die das objektorientierte Programmieren und die zugrunde liegenden Konzepte anhand von elektrotechnischen Fragestellungen erlernen oder vertiefen wollen.

Ich danke allen Personen, die zu diesem Buch beigetragen haben, für ihre kreativen Anregungen und Vorschläge. Ein ganz besonderer Dank gebührt Dr. Malte Probst, der das Manuskript Korrektur gelesen und an entscheidenden Stellen in die vorliegende Form gebracht hat.

Gießen, im April 2023 *Uwe Probst*

Sie finden die ausführlichen Lösungsvorschläge zu den Übungsaufgaben sowie die kompilierfähigen Projekte zu den Programmierübungen unter:

https://www.thm.de/ei/uwe-probst/lehrbuecher#vorlesung-softwareentwicklung

Inhalt

1 Einleitung

Viele Maschinen und Anlagen im Bereich der industriellen Fertigung weisen einen hohen Anteil an Software auf. Die dort eingesetzten Programme bilden einen wesentlichen Teil des Hersteller-Know-hows und sind eng mit dem elektrischen bzw. mechatronischen Gesamtsystem verknüpft. Der Konzeption, Erstellung und Pflege dieser Programme kommt damit eine hohe Bedeutung zu. Praktisch alle Ingenieure und Naturwissenschaftler, die mit der Entwicklung und dem Vertrieb technischer Anlagen befasst sind, müssen daher ausreichende Programmierkenntnisse besitzen. Dies gilt auch dann, wenn sie nicht selbst programmieren, sondern Entwicklungsprozesse vorwiegend koordinieren. In diesem Fall sind grundlegende Kenntnisse der Softwareentwicklungsprozesse und der ihnen zugrunde liegenden Methoden und Konzepte unerlässlich.

Dieses Buch richtet sich daher an Studierende der Ingenieur- und Naturwissenschaften und an Absolventen, die bereits im Berufsleben stehen. Nach der Einleitung ist der Einstieg – abhängig von den individuellen Vorkenntnissen – an unterschiedlichen Stellen sinnvoll. Lesenden, die vorwiegend Erfahrungen mit prozeduraler Programmierung haben, wird ein Beginn mit Kapitel 2 empfohlen. Sofern bereits grundlegende Kenntnisse der objektorientierten Programmierung vorhanden sind, können die Kapitel 2 und 3 zunächst übersprungen und mit Kapitel 4 begonnen werden.

In Kapitel 2 werden wichtige nicht objektorientierte Erweiterungen von C zu C++ erläutert. Hierunter fallen sowohl Neuerungen für Funktionen unter den Stichworten variable Parameterliste, Überladen, Ausnahmebehandlung (Exceptions) und *inline* als auch der neue Datentyp *bool* sowie Referenzen, die für den effizienten Datenaustausch zwischen Funktionen große Bedeutung gewonnen haben.

Ein wesentliches Ziel der modernen objektorientierten Programmierung (OOP) ist es, eine gute Wiederverwendbarkeit bereits vorhandener Programmteile zu erreichen und zu unterstützen. Dies erfordert die Trennung des Programmcodes in *Schnittstelle* und *Implementierung*. Hauptthema in Kapitel 3 ist daher die Erläuterung der beiden wichtigen OOP-Paradigmen Geheimnisprinzip und Vererbung.

Kapitel 4 ist der dynamischen Speicherverwaltung gewidmet. Sie ermöglicht es, zur Programmlaufzeit eine beliebige Anzahl neuer Objekte anzulegen und ist damit essentiell für viele Anwendungen. Neben der Beschreibung der Vorgehensweise werden Datenstrukturen besprochen, die den Umgang mit solchen Objekten erleichtern. In diesem Zusammenhang wird auch das Thema Vielgestaltigkeit (Polymorphie) an Beispielen erörtert.

Um das Vorgehen zu strukturieren, wird die Softwareentwicklung in einzelne Phasen gegliedert. Zur Beschreibung des Ablaufs werden sogenannte Vorgehensmodelle verwendet, die die zeitliche Abfolge der einzelnen Phasen festlegen.

Moderne, leistungsfähige Softwareprodukte und die dafür erforderlichen Entwicklungsprozesse müssen einer ganzen Reihe von Anforderungen gerecht werden, um technisch und ökonomisch wettbewerbsfähig zu sein. Betrachtet man den Aufbau moderner Anwendungen eingehend, stellt man fest, dass vergleichbare Produkte oft einen ähnlichen Aufbau haben, der sich bewährt hat. So sind PKWs meist vierrädrig, haben zwischen 5 und 7 Sitze und eine Motorleistung bis zu 140 kW. Nahezu alle Einfamilienhäuser weisen – unabhängig vom jeweiligen Architektenentwurf – neben Wohn-, Ess- und Schlafzimmer noch zwei bis drei Kinderzimmer, Küche und Bad auf. Auf die Softwareentwicklung werden diese Prinzipien unter dem Stichwort Entwurfsmuster übertragen. Hierbei handelt es sich um etwa 25 unterschiedliche Softwarearchitekturen. Jede dieser Lösungen ist auf ganz bestimmte Anwendungen zugeschnitten. Unter dem Stichwort Softwaretechnik werden verschiedene Vorgehensmodelle und einige Entwurfsmuster in Kapitel 5 besprochen.

Um die Anwenderwünsche während der Analysephase eines Softwareprojekts möglichst präzise zu erfassen, ist eine exakte Beschreibung der zu implementierenden Module unerlässlich. Sie stellt gewissermaßen das Fundament des späteren Programms bereit. In der Praxis verwendet man hierzu die moderne Beschreibungssprache UML, die anhand von Beispielen erläutert wird. Dies erfolgt nicht in einem separaten Kapitel, sondern immer dann, wenn neue Sprachelemente eingeführt werden. Auf diese Weise werden innerhalb der einzelnen Kapitel wesentliche Diagrammtypen der UML vorgestellt und exemplarisch angewendet.

Mithilfe der prozeduralen Programmierung können die wesentlichen genannten Forderungen aufgrund systematischer Einschränkungen nicht konsequent umgesetzt werden. Möglich wird dies mit dem objektorientierten Ansatz, der sowohl beim Erfassen der Anforderungen als auch in der Entwurfs- und Implementierungsphase zum Einsatz kommt. Ein unverzichtbares Hilfsmittel stellen fertige objektorientierte Bibliotheken dar, die häufig auf Vorlagen und Schablonen basieren. Beispiele für solche Bibliotheken sind die C++-Standardbibliothek (STL) sowie die Microsoft Foundation Classes (MFC), die beide in Kapitel 6 besprochen werden. Letztere bildet eine mögliche Grundlage zur Erstellung grafischer Oberflächen.

Den Abschluss bilden ingenieurwissenschaftliche Programmierbeispiele in Kapitel 7.

Die wesentlichen Sprachelemente der Programmiersprache C werden für das Verständnis des Buches vorausgesetzt (z. B. Arrays, Zeiger, Parameterübergabe bei Funktionen, Schleifen und Kontrollstrukturen). Alle Beispielprogramme wurden mit Visual Studio 2022 entwickelt und getestet. Auf der Webseite zum Buch können diese Beispiele einzeln heruntergeladen werden. Grundsätzlich sollten die Quellcodes der *.cpp-Dateien – mit Ausnahme der Beispiele in den Kapiteln 6 und 7, die herstellerspezifische Bibliotheken nutzen – auch mit anderen Compilern erfolgreich übersetzt werden können.

Die dargestellten Beispiele sollen jeweils gewisse Aspekte verdeutlichen und erheben nicht den Anspruch von Robustheit und Zuverlässigkeit. Man kann alles anders und besser machen.

2 Von C zu C++

Im Vergleich zur Sprache C enthält C++ eine ganze Reihe von Verbesserungen und Weiterentwicklungen, die nicht unmittelbar mit den objektorientierten Programmkonzepten zusammenhängen. Eine Auswahl dieser Erweiterungen wird nachfolgend besprochen.

2.1 Neues zu Funktionen

Lernziele
Der/die Lernende

- nutzt eine variable Parameterliste zur Vereinfachung der Schnittstelle,
- wendet die Technik des Überladens auf Funktionen an,
- erkennt die Vorteile von Funktionsschablonen.

2.1.1 Funktionen mit variabler Parameterliste

In C++ ist es möglich, dass man für einige der Funktionsparameter Standardwerte vorgibt. Werden die Parameter beim Aufruf nicht explizit vorgegeben, verwendet die Funktion automatisch die Standardwerte. Parameter dieser Art müssen am Ende der Parameterliste platziert werden:

```
void inkrement(int *x, int step=1) {
    *x += step;
}
void main(void) {
    int wert = 0;
    inkrement (&wert, 2);          // individuelle Angabe der Schrittweite: 2
    inkrement(&wert);              // Schrittweite nicht angegeben: 1
}
```

Die Funktion *inkrement(int *x, int step=1)* verwendet zwei Parameter. Der Parameter *step* wird mit dem Wert 1 vorbelegt. Der erste Aufruf von *inkrement* setzt *step* auf den Wert 2. Beim zweiten Aufruf wird dagegen nur der Zeiger übergeben. Da *step* nicht spezifiziert wurde, wird hier mit dem Standardwert 1 gearbeitet.

Standardwerte für Funktionsparameter werden mittels Zuweisungsoperator („=“) den formalen Parametern nachgestellt. Dabei gilt, dass nach einem Parameter mit Standardwert nur noch Parameter folgen dürfen, welchen ebenfalls ein Standardwert zugewiesen ist.

2.1.2 Überladen

Nach Möglichkeit sollten sprechende Funktionsnamen vergeben werden, so dass aus dem Namen hervorgeht, was die Funktion leistet. Hin und wieder muss die gleiche Funktion mit verschiedenen Datentypen arbeiten; allein deswegen musste in C der Name der Funktion angepasst werden.

Beispiel 2.1 Bestimmen des größten Wertes in einem *int*-Feld

Schreiben Sie die Funktion *int maxFinden(int feld[], int laenge)*, die den Wert des größten Feldelements als Rückgabewert liefert.

Lösung:

```
int maxFinden(int feld[], int laenge) {
    int max = feld[0];
    for (int i = 0; i < laenge; i++) {
       if (feld[i] > max)
                max = feld[i];
    }
    return max;
}
```

Sofern eine weitere Funktion zusätzlich zu der aus Beispiel 2.1 erforderlich wird, die z. B. ein Feld von *double*-Variablen durchsucht, wird die Technik des Überladens eingesetzt: Es wird eine Funktion gleichen Namens erstellt, die sich in der Parameterliste von der bereits vorhandenen Funktion unterscheidet.

Beispiel 2.2 Bestimmen des größten Wertes in einem *double*-Feld

Schreiben Sie die Funktion *double maxFinden(double feld[], int laenge)*, das den Wert des größten Feldelements als Rückgabewert liefert.

Lösung:

```
double maxFinden(double feld[], int laenge) {
    double max = feld[0];
    for (int i = 0; i < laenge; i++) {
       if (feld[i] > max)
                max = feld[i];
    }
    return max;
}
```

In nachfolgendem Programmfragment kann der Compiler sehr wohl unterscheiden, dass an der Stelle // 1 die Funktion *maxFinden* für *int*-Felder und bei // 2 eine Funktion gleichen Namens, aber für Felder vom Typ *double* aufgerufen werden soll.

```
int zahlenfeld[10] = {1,6,2,9,-12,5,23,45,-45,14};
cout <<endl <<"Max. Wert = " <<maxFinden(zahlenfeld, 10);     // 1
double wertefeld[10] ={3.1,-6.3,5.7,12.9,5.78,-3.56,23.9,3.3,6.5,1.2};
cout <<endl <<"Max. Wert = " <<maxFinden(wertefeld, 10);      // 2
```

Unter C++ werden Funktionen nicht allein über den Funktionsnamen erkannt, sondern durch die eindeutige Kombination von Funktionsname und Anzahl sowie Datentyp der formalen Parameter der Parameterliste. Man spricht in diesem Zusammenhang vom Überladen (overloading) von Funktionsnamen.

Das Überladen von Funktionen kann mit mehrdeutigen Begriffen verglichen werden. Die jeweilige Bedeutung von umgangssprachlichen Homonymen erschließt sich aus dem Kontext, in dem die Begriffe verwendet werden: Aus der Aussage „ich setze mich auf die Bank“ geht klar hervor, dass die Parkbank und nicht etwa ein Geldinstitut gemeint ist.

Ähnlich wird bei überladenen Funktionen aus dem „Sinnzusammenhang“, d. h. aus Anzahl und Datentyp der übergebenen Parameter, darauf geschlossen, welche der vorliegenden gleichnamigen Funktionen aufgerufen werden soll.

2.1.3 Inline-Funktionen

Beim Aufruf einer Funktion müssen eine ganze Reihe von Maßnahmen durchgeführt werden:

- Kopieren der Aufrufparameter und der Rücksprungadresse auf den Stack
- Ausführen des Unterprogrammaufrufs
- nach Ende der Funktion Sprung zur Rücksprungadresse, die auf dem Stack liegt
- ggf. Lesen des Rückgabewerts vom Stack
- Freigabe des für den Funktionsaufruf verwendeten Stackbereichs

Dieser Aufwand ist für kurze Funktionen häufig höher als die Ausführung des eigentlichen Funktionscodes. In C++ gibt es die Möglichkeit, Funktionen als *inline* zu deklarieren. Der Compiler sollte bei *inline*-Funktionen den Funktionscode an die Stelle des Aufrufs kopieren, so dass kein Funktionsaufruf mehr benötigt wird. Die Effizienz einer *inline*-Funktion entspricht dann der Effizienz eines Makroaufrufs.

Listing 2.1 Definition einer *inline*-Funktion

```
inline int max(int x, int y) {
    return (x>y?x:y);
}
```

Allerdings ist die Angabe von *inline* lediglich ein Hinweis an den Compiler, den Funktionscode zu kopieren. Er entscheidet, ob solch eine Funktion mittels normalem Funktionsaufruf oder als *inline* implementiert wird.

2.2 Referenzen

Lernziele

Der/die Lernende

- unterscheidet Referenzen und Zeiger,
- wendet Referenzen zur Parameterübergabe an Funktionen an.

2.2.1 Definition

Während man in C auf Variablen entweder direkt über ihren Namen oder indirekt über Zeiger zugreifen kann, gibt es in C++ mit den Referenzen eine dritte Möglichkeit. Referenzen stellen einen zweiten Zugang, einen alternativen Namen, für die Nutzung von Variablen zur Verfügung. Wichtigstes Einsatzgebiet für Referenzen sind die Parameterübergabe bei Funktionsaufrufen und die Rückgabe berechneter Werte durch Funktionen.

Beispiel 2.3 Vertauschen zweier Werte

Gebräuchliche Sortieralgorithmen verwenden an zentraler Stelle das Vertauschen zweier Variablenwerte. Schreiben Sie eine Funktion *vertausche(int _x, int _y)*, die die Werte der als Parameter übergebenen Variablen miteinander vertauscht.

```
void main(void) {
    int a = 3, b = 5;
    cout <<"a: " << a <<" b: " <<b <<endl;
    vertausche(a, b);
    cout <<"a: " << a <<" b: " <<b <<endl;
}
```

Lösungsversuch 1:

```
void vertausche(int _x, int _y) {
    int zw = _y;
    _y = _x;
    _x = zw;
}
```

Der erfahrene Programmierer versteht sofort, dass die vorliegende Lösung mit *call by value* arbeitet und nicht korrekt funktionieren kann: Bei dieser Wertübergabe werden Kopien der Variablen *a* und *b* an die Funktion *vertausche* übergeben. Der in der Funktion ablaufende Algorithmus beeinflusst nur die Kopien, nicht aber die Originalwerte von *a* und *b*. Das Vertauschen ist nicht erfolgreich.

Lösungsversuch 2:

Um die Aufgabe erfolgreich anzugehen, muss mit einer Parameterübergabe *call by reference* gearbeitet werden. Dies gelingt durch Verwendung von Zeigern:

```
void vertausche(int * _x, int * _y) {
    int zw = * _y;
    * _y = * _x;
    * _x = zw;
}
```

Diese Lösung funktioniert, wenn der Funktionsaufruf *vertausche(a, b)* im obigen Programmfragment mit Zeigern formuliert und durch *vertausche (&a, &b)* ersetzt wird. Allerdings ist die Programmierung mithilfe von Zeigern aufgrund der indirekten Adressierung komplex und daher fehleranfällig.

Durch den Einsatz von Referenzen gelingt es, eine Parameterübergabe der Art *call by reference* zu programmieren, ohne dass Zeiger verwendet werden müssen.

Die Definition von Referenzen erfolgt mit dem &-Zeichen. Referenzen müssen – im Gegensatz zu anderen Variablen – unmittelbar bei der Definition initialisiert werden. Auf das referenzierte Objekt kann im Anschluss sowohl über seinen ursprünglichen Namen als auch über die Referenz in gleicher Art und Weise zugegriffen werden:

```
int zahl = 15;            // Variable zahl vereinbaren
int &rZahl = zahl;        // rZahl ist eine Referenz auf zahl
int x = rZahl;            // x erhält den Wert von zahl
rZahl = 2;                // zahl wird (über die Referenz) der Wert 2 zugewiesen
```

Der Umgang mit Referenzen ist i. A. einfacher als der mit Zeigern. Zwischen den beiden Zugängen existieren zwei wesentliche Unterschiede, wie anhand des nachfolgenden Fragments deutlich wird:

```
int y, z;
int &refAufY = y;
int * ptr = &y;
ptr = &z;                 // 1
ptr++;                    // 2
*ptr = 5;                 // 3
refAufY = 18;             // 4
```

- Der Zeiger *ptr* kann mit // 1 bzw. // 2 zur Laufzeit des Programms noch verändert werden, die Referenz *refAufY* jedoch nicht!
- Beim Zeiger wird aufgrund des Zugriffs // 3 sofort ersichtlich, dass es sich um einen Verweis auf diejenige Speicherzelle handelt, auf die *ptr* zeigt. Bei Verwendung der Referenz in // 4 ist nicht sofort deutlich, dass *y* den Wert 18 zugewiesen bekommt.

Eine Referenz muss unmittelbar bei ihrer Definition initialisiert werden. Sie ist im Prinzip ein konstanter Zeiger auf eine andere Variable. Die Bindung der Referenz an die Variable kann nach der Initialisierung nicht mehr gelöst oder verändert werden.

2.2.2 Referenzen und Funktionen

Referenzen als Funktionsparameter

Liefert eine Funktion einen einzelnen Rückgabewert, so wird dieser mittels der *return*-Anweisung identifiziert. Funktionen, die mehr als einen Rückgabewert berechnen, müssen die Rückgabe über die Funktionsparameter vornehmen. Dies bedeutet, dass die zu verändernden Variablen der Funktion mit *call by reference* als Parameter übergeben werden müssen.

Dies gelingt in der Sprache C nur, wenn die Parameterübergabe mittels Zeigern realisiert wird.

Übung 2.1

Schreiben Sie eine Funktion, die die Lösung einer quadratischen Gleichung mithilfe der pq-Formel ermittelt:

$$0 = x^2 + p \cdot x + q \qquad \Rightarrow \qquad x_{1,2} = -\frac{p}{2} \pm \frac{1}{2} \cdot \sqrt{p^2 - 4 \cdot q}$$

Übung 2.2

Gegeben ist die Funktion *reziprok(double& _r)*, die den Parameter *_r* durch seinen Kehrwert ersetzt. Der Parameter *_r* wird als Referenz übergeben.

```
void reziprok(double& _r) {
    _r = 1./_r;
}
```

Beurteilen Sie, welche Ergebnisse von nachfolgendem Programmfragment berechnet werden und welche Aufrufe zulässig sind.

```
#include <iostream>
void main() {
    double x = 0.25;
    reziprok(x);
    cout << x;
    reziprok(0.2);
}
```

Referenzen als Rückgabewerte

Referenzen können auch als Rückgabewerte von Funktionen verwendet werden. Dies lohnt sich dann, wenn die Rückgabewerte große Objekte sind, deren Kopieren zeitaufwendig wäre. Allerdings ist Vorsicht geboten: Die Rückgabe von Referenzen auf lokale Variablen von Funktionen ist nach Beispiel 2.4 gefährlich.

Beispiel 2.4 Rückgabe von Referenzen auf lokale Variablen

Schreiben Sie die Funktion *double & betragBerechnen(Complex _c)*, die den Betrag der als Parameter übergebenen komplexen Zahl *_c* berechnet.

Lösung:

```
double & betragBerechnen(Complex _c) {
    double betrag;       // Lokale Variable anlegen
    betrag = sqrt(_c.real * _c.real + _c.imag * _c.imag);
    return betrag;       // Referenz auf lokale Variable zurückliefern
}
```

Die Variable *betrag* wird auf dem Stack angelegt; nach dem Ende der Funktion *betragBerechnen()* wird sie vom Stack gelöscht und besitzt daher keine Gültigkeit mehr. Die Variable *betragVonC1* des Hauptprogramms erhält somit eine Referenz auf ein **ungültiges** Objekt.

```
#include <iostream>
using namespace std;
void main() {
    Complex c1 = {3, 4};          // c1 wird auf dem Stack angelegt
    double &betragVonC1 = betragBerechnen(c1);
    cout <<endl << "|c1|: ";      // verändert den Stack dort, wo die
                                  // lokale Variable betrag gelegen hat
    cout << betragVonC1<< endl;   // Ausgabe des falschen Wertes
}
```

Im Hauptprogramm wird die Referenz auf das lokale Objekt *betrag* in der Referenz *betragVonC1* gespeichert. Die nachfolgende Ausgabe der Zeichenkette verwendet den Stack und überschreibt dabei das jetzt ungültige Objekt *betrag*. Die sich anschließende Ausgabe des ungültigen Objektwerts mittels der Referenz *betragVonC1* wird mit hoher Wahrscheinlichkeit nicht den korrekten Wert 5 ausgeben.

Das vollständige Programm „Referenzen“ finden Sie unter
https://www.thm.de/ei/uwe-probst/lehrbuecher#vorlesung-softwareentwicklung

Ein guter Compiler wird beim Übersetzen der Funktion *betragBerechnen()* vor der Rückgabe einer Referenz auf eine lokale Variable warnen. Bei sorgfältiger Beachtung der Compiler-Warnungen können solche Fehler frühzeitig erkannt werden. Im Zusammenhang mit Klassen werden Referenzen im Kapitel 3 weiter vertieft werden.

2.3 Datentyp *bool*

Für boolesche Variablen steht in C++ der eingebaute Datentyp *bool* zur Verfügung. Eine Variable dieses Typs speichert einen Wahrheitswert, der entweder richtig *(true)* oder falsch *(false)* sein kann. In C++ sind *true* und *false* Schlüsselbegriffe.

Beispiel 2.5 Anwendung von Variablen des Typs bool

Das nachfolgende Programmfragment illustriert mögliche Zuweisungen an *bool*-Variablen. Neben direkten Zuweisungen ist auch die Berechnung von logischen Ausdrücken zulässig.

```
bool a,b,c;
int i;
// mögliche Zuweisungen von true
a = true;
a = 17;          // a wird zu true
a = (1<2);       // der log. Ausdruck 1 < 2 ist true, damit wird a true
// mögliche Zuweisungen von false
b = false;
b = 0;           // b wird zu false
b = (1>2);       // der log. Ausdruck 1 < 2 ist false, damit wird b false
c = a & b;       // c ist false, da b == false und a == true
i = a;           // i wird zu 1
i = c;           // i wird zu 0
```

Aus Beispiel 2.5 leitet man folgende Regeln für die Verwendung von *bool*-Variablen ab:

- Wird ein *int*-Wert einer *bool*-Variablen zugewiesen, so erhält sie den Wert *true*, sofern die *int*-Variable einen Wert ungleich 0 hat. Ist der *int*-Wert 0, so erhält die *bool*-Variable den Wert *false*.
- Wird ein *bool*-Wert einer *int*-Variablen zugewiesen, so erhält diese bei *false* den Wert 0 und bei *true* den Wert 1.
- In arithmetischen und logischen Ausdrücken wird implizit ein *bool*-Wert in einen *int*-Wert konvertiert.
- Wird das Ergebnis eines arithmetischen Ausdrucks einer *bool*-Variablen zugewiesen, erfolgt implizit die Konvertierung von *int* nach *bool*.

2.4 Namensbereiche

Namensbereiche (name spaces) vermeiden Konflikte bei globalen Objekten bzw. Funktionen. Ein Namensbereich wird durch das Schlüsselwort *namespace* sowie einen nachfolgenden Bezeichner eingeleitet. Es folgt ein Block mit Definitionen, die dem Namensbereich zugeordnet werden.

Beispiel 2.6 Namensbereiche

In diesem Beispiel werden die beiden Namensbereiche *autor1* und *autor2* angelegt. In jedem der Namensbereiche und auch außerhalb eines speziellen Namensbereiches werden gleichnamige Variablen und Funktionen definiert.

```
namespace autor1{         // Namensbereich "autor1"
    int wert = -28;
    void ausgabe (int _w){
       cout <<endl <<"Die Ausgabe lautet: " <<_w;
    }
}
namespace autor2{         // Namensbereich "autor2"
    double wert = 12;
    void ausgabe (double _w){
       cout <<endl <<"Wert: " <<_w;
    }
}
int wert = 96;            // außerhalb eines speziellen Namensbereiches
void ausgabe (int _wert){
    cout <<endl <<"Ausgabewert: " <<_wert;
}
```

Im Hauptprogramm muss der Compiler wissen, welche der drei Funktionen namens *ausgabe* jeweils gemeint ist. Dies geschieht durch Angabe des Namensraumes, der dem Funktionsnamen vorangestellt und durch den '::'-Operator von ihm getrennt wird. Die Funktion, die außerhalb eines speziellen Namensraumes definiert wurde, wird wie gewohnt aufgerufen.

```
void main(void) {
    int zahl = 34;
    autor1::ausgabe(zahl);                // 1 Die Ausgabe lautet: 34
    autor2::ausgabe(autor1::wert);        // 2 Wert: -28
    autor1::ausgabe(autor2::wert);        // 3 Die Ausgabe lautet: 12
    ausgabe(wert);                        // 4 Ausgabewert: 96
}
```

Bei // 1 wird die Funktion aus Namensbereich *autor1* mit der lokalen Variable *zahl* als Parameter aktiviert. Unter // 2 wird *ausgabe* aus *autor2* mit der Variable *wert* aus Namensbereich *autor1* verwendet. Im Fall // 3 kommt wieder *ausgabe* aus Namensbereich *autor1* zum Einsatz, diesmal jedoch mit der Variable *wert* aus dem Namensbereich *autor2*. Zuletzt wird bei // 4 die Funktion *ausgabe* mit der globalen Variable *wert* benutzt.

In Beispiel 2.6 erfolgen zwar gleichnamige Definitionen von Variablen und Funktionen, allerdings in unterschiedlichen Namensbereichen. Dadurch ist die Verwechslungsgefahr mit gleichnamigen Definitionen in anderen oder außerhalb von Namensbereichen gering.

Ein Namensbereich muss nicht in einem einzelnen Block vereinbart werden, sondern lässt sich bei Bedarf erweitern: Wenn die nachfolgende Anweisung hinter der Definition des Namensbereichs *autor2* angeordnet wird, fügt sie dem Namensbereich *autor1* die Variable *fehler* hinzu:

```
namespace autor1{
    int fehler;
}
```

Gelegentlich ist es nützlich, den gesamten Namensbereich mit der *using*-Direktive zu deklarieren:

```
using namespace autor1;
// ab hier können alle Objekte des Namensbereiches autor1
// normal verwendet werden
```

Dieses Vorgehen wird häufig für die Ein-/Ausgabefunktionalitäten benutzt, die sämtlich im Namensbereich *std* definiert worden sind.

2.5 Ein- und Ausgabe

Lernziele
Der/die Lernende

- versteht den Begriff des Ein- und Ausgabestroms,
- wendet die Mechanismen der objektorientierten Ein- und Ausgabe an,
- überträgt diese Mechanismen auf den Umgang mit Dateien.

C++ verwendet zur Ein-/Ausgabe einheitliche Ströme von Zeichen – die sogenannten Streams. Sie bilden die Schnittstelle zwischen dem Prozessor und seiner Peripherie und ermöglichen den Datenaustausch mit Bildschirm, Tastatur, Drucker, Scanner, Datei und Netzwerk.

Die Basis für alle Ein- und Ausgabeoperationen bilden Datenströme von Zeichen. Auf der untersten Ebene wird ein solcher Stream als Folge von Bytes aufgefasst: Von einem Input-Stream werden einzelne Zeichen eines Eingabegerätes eingelesen. Auf einen Output-Stream können Daten Zeichen für Zeichen ausgegeben werden. Diese Methoden sind universell auf allen Plattformen verfügbar. Ein Programm, das nur diese Ein- und Ausgabetechnik benutzt, läuft demnach auf verschiedenen Rechnerarchitekturen oder Betriebssystemen.

Die *iostream*-Bibliothek stellt die erforderlichen Ein- und Ausgabefunktionen für jeden Standard-Datentyp zur Verfügung.

2.5.1 Standardausgabe

Die Standardausgabe erfolgt in C++ über einen Stream, der durch den Namen *cout* vordefiniert ist. Ausgaben über den Stream nutzen den Operator <<, der Ausgabeoperator genannt wird und nicht mit dem „Schieben nach links“ verwechselt werden darf.

Die unterschiedliche Funktionalität zwischen Schiebeoperationen und Ausgeben wird durch Überladen des Operators für Stream-Datentypen erreicht. Eine typische Bildschirmausgabe mithilfe des Streams *cout* zeigt Beispiel 2.7.

Beispiel 2.7 Ausgabe mit *cout*

Eine Ausgabe unter C++ erfolgt mithilfe des Operators << und der Angabe des Stroms. Die Bibliothek *iostream* muss eingebunden werden.

```
#include <iostream>       // iostream-Bibliothek einbinden
using namespace std;      // Namensbereich std verwenden
void main(void) {
    cout << "Hello World\n";
}
```

Über einen Stream können neben Texten auch alle Standard-Datentypen von C++ ausgegeben werden. Das lästige Formatieren der Ausgabe, das bei *printf()* erforderlich war, entfällt, da der Operator << für die Standard-Datentypen überladen ist.

Soll etwa das Quadrat des Wertes einer Variablen k ausgegeben werden, wird dies folgendermaßen erreicht:

```
int k = 13;
cout << "Das Quadrat von";
cout << k;
cout << "ist";
cout << k*k;              // der Ausdruck k*k wird zunächst berechnet
                          // und das Ergebnis dann ausgegeben
cout << "\n";
```

Der Ausgabeoperator wird von links nach rechts abgearbeitet. Daher kann die Ausgabe in einer Zeile und zu einer Anweisung zusammengefasst werden.

```
int k = 13;
cout << "Das Quadrat von" << k << "ist" << k*k << "\n";
```

Formatierung der Ausgabe

Zur Formatierung der Ausgabe werden Manipulatoren verwendet, die in den Ausgabestrom eingefügt werden. Sie beeinflussen den Ausgabestrom entweder dauerhaft oder nur für die unmittelbar folgende Ausgabe. In gleicher Weise können sie ebenso für Dateiströme benutzt werden. Eine Übersicht über einige Manipulatoren wird in Tabelle 2.1 gegeben. Weitergehende Informationen finden sich in [Lang03], [Erl04].

Tabelle 2.1 Manipulatoren für Ausgabeströme; die mit 1) gekennzeichneten Manipulatoren erfordern die Einbindung von <iomanip>

Manipulator	Gültigkeit	Bedeutung
flush	–	Ausgabedaten sofort ausgeben (Ausgabepuffer leeren)
endl	–	'\n' ausgeben mit anschließendem flush
dec	permanent	dezimale Zahlendarstellung
hex	permanent	hexadezimale Zahlendarstellung
setbase(int b) [1)]	permanent	Integerwert zur Basis b ausgeben. Gültige Werte für b sind: 8, 10, 16
setw(int n) [1)]	nächste Ausgabe	bestimmt die minimale Breite des Ausgabefeldes
setfill(int c) [1)]	nächste Ausgabe	legt Füllzeichen zum Auffüllen des Ausgabefeldes fest
setprecision(int n) [1)]	permanent	Anzahl der gültigen Ziffern bei der Ausgabe von Gleitkommawerten

Beispiel 2.8

Geben Sie an, welche Ausgabe durch nachfolgendes Programmfragment erzeugt wird.

```
#include <iostream>
#include <iomanip>
void main(void) {
     cout << "1: " << dec << 17 << endl;
     cout << "2: " << hex << 17 << endl;
     cout << "3: " << setbase(10) << 17 << endl;
     cout << "4: " << setw(10) << setfill('-') << "Hallo" << endl;
     cout << "5: " << 1.1234567890123 << endl;
     cout << "6: " << setprecision(10) << 1.1234567890123 << endl;
}
```

Lösung:

Das Programm führt zur nachfolgenden Ausgabe. An Position 5 erfolgt die Ausgabe mit der Standardbreite von 6 Nachkommastellen inkl. Dezimalpunkt. Diese wird an Position 6 durch den Manipulator *setprecision(10)* auf die Breite 10 geändert.

```
1: 17
2: 11
3: 17
4: -----Hallo
5: 1.12346
6: 1.123456789
```

2.5.2 Standardeingabe

Als Standardkanal zum Einlesen von Daten wird der Stream *cin* eingesetzt. Das Einlesen erfolgt mit dem Operator >>, der Eingabeoperator genannt wird. Auch dieser Operator ist für *stream*-Datentypen überladen und hat hier ebenfalls nichts mit einer Schiebeoperation zu tun.

Folgende Anweisung liest einen Datenwert aus der Standardeingabe und legt den Wert in der Variablen *m* ab:

```
int m;
cin >> m;
```

Wie schon beim Ausgabeoperator können mehrere Eingabeoperationen in einer Anweisung zusammengefasst werden.

```
double a;
int b;
cin >> a >> b;
```

Mit den Manipulatoren *dec* und *hex* aus Tabelle 2.1 kann die Basis, zu der die Zahlen interpretiert werden, auch bei der Eingabe umgeschaltet werden.

Beispiel 2.9

Erläutern Sie, wie die Werte für *a* und *b* in nachfolgendem Programm eingegeben werden müssen.

```
#include <iostream>
void main(void) {
    int a,b;
    cin >> a >> hex >> b;
    cout << "a=" << a << ", b=" << b << endl;
}
```

Lösung:

Der Wert für *a* muss in dezimaler Darstellung eingegeben werden. Unmittelbar danach wird mittels des Manipulators *hex* auf hexadezimale Eingabe umgeschaltet. Daher wird der Wert für *b* als hexadezimale Eingabe erwartet.

2.5.3 Lesen und Schreiben von Dateien

Die Stream-Funktionalität wird auch beim Lesen aus oder Schreiben in Dateien genutzt. Standardmäßig werden die Stream-Typen *ifstream* (Lesen) und *ofstream* (Schreiben) bereitgestellt. Die Verwendung beider Typen setzt das Einbinden von <*fstream*> voraus.

Zunächst wird durch das Anlegen einer Variablen ein Stream für die betreffende Datei erzeugt. Bei ihrer Definition wird dieser Variablen der Name der Datei übergeben.[1]

In der Praxis muss natürlich u. a. überprüft werden, ob die Datei gefunden und geöffnet werden konnte. Nur dann sind sinnvolle Leseoperationen möglich. Während des Lesens erfolgt die Kontrolle, ob das Dateiende erreicht wurde. Dazu bieten alle Streams (auch der Standard-Eingabestrom *cin*) verschiedene Möglichkeiten:

- Mit der Methode *eof()* kann der Eingabestatus während der letzten Eingabeoperation abgefragt werden. Die Methode liefert den Wert 0, wenn weiter gelesen werden kann. Wurde bei der letzten Leseoperation das Dateiende erreicht, gibt sie einen Wert ungleich 0 zurück.
- Mit der Methode *good()* wird festgestellt, ob das Öffnen der Datei erfolgreich war.

Beispiel 2.10 Lesen aus einer Datei

Schreiben Sie ein Programmfragment, das die Datei „Log.txt“ zum Lesen öffnet und den Inhalt der Datei auf den Bildschirm ausgibt.

Lösung:

Zunächst wird die Variable *inFile* angelegt, die den Datentyp des gewünschten Streams hat. Die Datei „Log.txt“ soll zum Lesen geöffnet werden, daher wird der Datentyp *ifstream* verwendet. Beim Anlegen der Variablen *inFile* wird der Name der zu öffnenden Datei übergeben.

Das Einlesen von Werten aus der Datei erfolgt mithilfe des Eingabeoperators. Quelle des Datenstroms ist jetzt allerdings nicht mehr der Standardeingabestrom *cin*, sondern eben der Stream aus der geöffneten Datei. Damit ergibt sich folgendes Programmfragment:

```
#include <fstream>                // Einbinden der Stream-Bibliothek
#include <iostream>
using namespace std;
void main(void) {
    char c;
    ifstream inFile("log.txt");   // Anlegen der Stream-Variablen und
                                  // verbinden mit der Datei log.txt, sofern
                                  // die Datei geöffnet werden konnte
    if (inFile.good()) {
       while (!inFile.eof()) {
                inFile >> c;      // Einlesen eines Zeichens aus der
                                  // Datei in die Variable c
                cout << c;        // Ausgabe des gelesenen Zeichens
       }
    } else
}
```

[1] Die Übergabe des Dateinamens beim Anlegen einer Stream-Variablen ist ein C++-Sprachkonstrukt, das in Abschnitt 3.2.4 unter dem Begriff Konstruktor erläutert wird.