

HANSER



Leseprobe

zu

Objektorientiertes Programmieren

von Uwe Probst

Print-ISBN: 978-3-446-47651-6

E-Book-ISBN: 978-3-446-47838-1

Weitere Informationen und Bestellungen unter

<https://www.hanser-kundencenter.de/fachbuch/artikel/9783446476516>

sowie im Buchhandel

© Carl Hanser Verlag, München

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 Programmier-techniken sowie der UML-Diagramme in diesen Themengebieten veranschaulichen.

Im fünften Kapitel stehen Programmier-techniken 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 herstellereigenen 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	Vorwort	5
1	Einleitung	11
2	Von C zu C++	13
2.1	Neues zu Funktionen	13
2.1.1	Funktionen mit variabler Parameterliste	13
2.1.2	Überladen	14
2.1.3	Inline-Funktionen	15
2.2	Referenzen	16
2.2.1	Definition	16
2.2.2	Referenzen und Funktionen	18
2.3	Datentyp <i>bool</i>	20
2.4	Namensbereiche	20
2.5	Ein- und Ausgabe	22
2.5.1	Standardausgabe	22
2.5.2	Standardeingabe	25
2.5.3	Lesen und Schreiben von Dateien	25
2.5.4	Ausgabe selbst definierter Datentypen	27
2.6	Exceptions und Fehlerbehandlung	28
2.6.1	Einführung	28
2.6.2	Ausnahmen	30
2.6.3	Verwendung vorhandener Exception-Klassen	32
2.7	Lösungen	37
3	Klassen und Objekte	39
3.1	Motivation zur Einführung der OOP	39
3.2	Von <i>struct</i> zu <i>class</i>	43
3.2.1	Klassendeklaration	43
3.2.2	Geheimnisprinzip: <i>private</i> und <i>public</i>	44
3.2.3	Objekte und Variablen	45
3.2.4	Konstruktor und Destruktor	45
3.2.5	Objekt: Instanz einer Klasse	50
3.2.6	Aufteilung in Dateien	53

3.2.7	Konstante Objekte	54
3.2.8	Hinweise zur Konstruktion von Methodenschnittstellen	55
3.2.9	<i>this</i> -Zeiger	56
3.2.10	Zusammengesetzte Klassen	57
3.2.11	Überladen von Operatoren	62
3.3	Vererbung	66
3.3.1	Gemeinsamkeiten und Unterschiede	66
3.3.2	Vererbungshierarchie	68
3.3.3	Ableitung und Zugriffsrechte	70
3.3.4	Initialisierung bei abgeleiteten Klassen	74
3.3.5	Abstrakte Klassen	74
3.4	Lösungen	75

4 Dynamische Speicherverwaltung 81

4.1	Dynamisches Anlegen von Objekten	81
4.2	Felder mit dynamisch änderbarer Länge	83
4.2.1	Einführende Beispiele	83
4.2.2	Unterschied zwischen tiefer und flacher Kopie	88
4.2.3	Initialisierung durch Kopie	90
4.3	Behälterklassen	91
4.3.1	Einfach verkettete Listen	91
4.3.2	Weitere Arten von Behälterklassen	99
4.4	Polymorphie am Beispiel von Listen	100
4.4.1	Einführung	100
4.4.2	Einfache Liste mit Bauelementen	101
4.4.3	Implizite Typumwandlungen in C++	103
4.4.4	Polymorphie	105
4.5	Lösungen	108

5 Techniken der Softwareentwicklung 113

5.1	Grundlagen	114
5.1.1	Fünf Phasen der Softwareentwicklung	114
5.1.2	Vorgehensmodelle	115
5.2	Einfache Entwurfsmuster	120
5.2.1	Einführung und Begriffe	120
5.2.2	Grundlagen von Entwurfsmustern	123
5.2.3	Strategie	125
5.2.4	Adapter	131
5.2.5	Beobachter	135
5.3	Multithreading	147
5.3.1	Grundlagen und Begriffe	147
5.3.2	Eigenschaften von Threads	152
5.3.3	Synchronisation von Threads	153
5.4	Lösungen	160

6	Klassenbibliotheken	167
6.1	Vorlagen, Schablonen, Templates	168
6.1.1	Makros	168
6.1.2	Templates	170
6.1.2.1	Funktionstemplates	170
6.1.2.2	Klassentemplates	173
6.1.2.3	Template-Metaprogrammierung	177
6.2	Wichtige Bestandteile der C++-Standardbibliothek	179
6.3	Microsoft Foundation Classes (MFC)	184
6.3.1	Grafische Benutzeroberflächen	184
6.3.2	Grundlegendes zu Windows-Programmen	184
6.3.3	Aufbau der MFC	190
6.3.4	Programmierung eines einfachen Taschenrechners	194
6.3.5	Grafische Ausgabe	202
6.4	Model-View-Controller	219
6.5	Lösungen	242
7	Beispielanwendungen	253
7.1	Visualisierung von Messwerten	254
7.1.1	Anwendungsfälle	254
7.1.2	Analyse	255
7.1.3	Entwurf	257
7.2	Erstellen von Bode-Diagrammen	265
7.2.1	Analyse	266
7.2.2	Entwurf	266
7.3	Lösungen	269
	Literatur	273
	Index	275

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 herstellereigene 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:

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



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.

Index

A

Ableitung 70
abstrakte Klasse 74, 103, 110, 127,
139
Adapterklasse 131
Aggregation 59, 77
Algorithmen 179, 181
Algorithmenfamilie 128
Analyse 114, 266
Assoziation 59
assoziative Container 180
Attribute 43, 49, 198, 208
Ausgabeoperator 22, 27
Ausnahmebehandlung (exception
handling) 30

B

Basisklasse 68, 72, 74, 80, 106, 127
Behälterklasse 91, 94, 100, 210, 216
Beobachter 137, 261
Beobachtermuster 137
Bode-Diagramm 266

C

C++-Standardbibliothek 179
call by reference 16, 18, 37, 55
call by value 16, 55, 91, 108
catch 30
cin 25
class 43
Containerklassen 99, 179, 181
cout 22

D

Default-Konstruktor 61
delete 82, 85
Deque 179
Destruktor 45, 49, 107
Device Context DC 202
doppelt verkettete Listen 99
dynamische Speicherverwaltung 81, 85,
92

E

Eigenschaften 43
Eingabeoperator 25
Entwurf 114, 266
Entwurfsmuster „Strategie“ 259
Ereignisse 185
evolutionäre Entwicklung 116
Exception 83
Exception-Klassen 32

F

FIFO-Prinzip 99
flache Kopie 89f.
freie Funktion 66
Frequenzgang 265f.
fstream 25
Funktionsparameter 13, 18
Funktionstemplate 168, 170

G

Geheimnisprinzip 45, 51, 66, 80
generische Programmierung 168
Gerätekontext 202f., 210
get-Methode 51

grafische Benutzeroberfläche 184
GUI-Klassen 184

H

Heap 81

I

ifstream 25
implizite Typumwandlungen 103
Initialisieren 49
Initialisierung 74
Initialisierung durch Kopie 90
Initialisierungsliste 74, 199
inkrementelle Entwicklung 117
Inline-Funktionen 15
Input-Stream 22
Installation/Wartung 115
Instanz 50
iostream 23
Iteratoren 179 f.

K

Kapselung 45, 126
Kardinalität 60 f., 77
Klassen 39, 192
Klassenassistent 208
Klassendefinition 53
Klassendeklaration 43, 53, 195
Klassendiagramm 49, 77
Klassenhierarchie 66, 68 f., 80
Klassentemplate 168, 170
Konstruktor 45 f., 61, 74, 195
Kopierkonstruktor 47 ff., 83, 90, 108

L

late binding 106
LIFO-Prinzip 99
Listenelement 93
lose Kopplung 135, 138, 145, 259

M

Makros 168
Manipulatoren 25
Mehrfachvererbung 69
Membervariablen 43
Message Map 200
Methode 43, 49, 51, 66, 75, 192, 200
Methodenschnittstellen 55, 83
MFC 212, 215
MFC-Bibliothek 190
Microsoft Foundation Classes (MFC) 184
modale Dialoge 213
mutex 153

N

Nachkomme 66
Nachrichten 52, 185 ff., 192, 200, 202, 208
Nachrichtenzuordnung 200
Namensbereich 20
new 82
next-Zeiger 93, 98

O

Oberklasse 66
Objektdiagramm 52
Objekte 39, 50
ofstream 25
Operator << 23
Operator >> 25
Operatorüberladung 63, 66
Output-Stream 22

P

Parameterliste 13, 49, 107
Parameterübergabe 27
Phasen der Softwareentwicklung 114
polymorphe Klasse 107, 110
polymorphe Methode 139
Polymorphie 100, 105
private 44 f., 49, 68, 75
Programmierschnittstelle 184

Programmierung 115
 protected 68
 Prozessmodelle 115
 public 44 f., 49, 68
 Publish-Subscribe-Prinzip 137
 pure virtual function 110

R

Referenz 16, 27, 38, 55
 RGB 206
 Rückgabetypp 49
 Rückgabewert 55, 83
 Rückgabewerte von Funktionen 18

S

Schablone 168, 170
 Schnittstelle 39, 45, 50, 69, 131
 Schnittstellenadapter 131
 schwache Aggregation 60
 Scope-Operator 48
 set-Methode 51
 Softwarequalität 69
 Speicherleck 88
 Spezialisierung 69, 72
 Stabilitätsanalyse 266
 Stack 81
 Standardausgabe 22
 Standardbibliothek 182
 Standardcontainer 179
 Standarddialoge 212
 Standardeingabe 25
 Standardkonstruktor 49, 74, 86
 Standard-Kopierkonstruktor 90
 Standard-Template-Library 168
 Stapelspeicher 99
 starke Aggregation 60
 Startzeiger 93
 statische Bindung 106
 statische Speicherverwaltung 81
 Steuerelemente 190, 192 f., 195, 198
 STL 179
 Strategie 161
 Strategiemuster 126
 Stream 22

struct 43
 Systemanalyse 265

T

Template 168, 170
 Test 115
 this-Zeiger 56
 Thread
 - mutex 153
 - Reihenfolgebedingung 156
 - Wechselseitiger Ausschluss 153
 throw 30
 tiefe Kopie 89 f.
 try 30
 Typ des Objektes 106
 Typ des Zeigers 106
 Typkonversionen 103
 Typumwandlungen 104

U

Überladen 14, 22, 27, 46, 62 f., 88 ff., 168
 überladener Konstruktor 86
 Überschreiben 72, 74
 Übertragungsfunktion 265
 Übertragungsglieder 265
 UML 60, 67, 254
 UML-Diagramm 61, 253, 260
 Unterklasse 66

V

Vektoren 179
 Vererbung 66 f., 71, 102
 Vererbungshierarchie 68, 103
 Vererbungsprinzip 66
 Verhaltensdiagramme 254
 Verhaltensmuster 126
 Verketteten von Operationen 90
 verkettete Listen 91
 verteilte Datenstrukturen 91
 Vielgestaltigkeit 100
 virtual 106
 virtuell 127
 virtuelle Methode 75, 107

Vorgehensmodelle 115
Vorlagen 168, 170

W

Warteschlange 99, 187
Wasserfallmodell 115, 117
Wiederverwendung 66

WinAPI 184f., 188, 190
WM_PAINT 211

Z

Zeiger 16, 55, 81, 83, 88, 92, 104, 106
Zugriffsrechte 70
Zuweisungsoperator 89f.