HANSER

Leseprobe

Uwe Probst

Objektorientiertes Programmieren für Ingenieure

Anwendungen und Beispiele in C++

ISBN (Buch): 978-3-446-44234-4

ISBN (E-Book): 978-3-446-44178-1

Weitere Informationen oder Bestellungen unter

http://www.hanser-fachbuch.de/978-3-446-44234-4

sowie im Buchhandel.

Vorwort

Inzwischen sind im Zuge der Umsetzung des Bologna-Prozesses nahezu alle Diplom-Studiengänge auf die neuen Bachelor- und Masterprogramme umgestellt worden. Durch die Neuordnung wurden die Präsenzphasen an den Hochschulen gekürzt und Studierende zu mehr Eigenarbeit veranlasst. Diese Eigenarbeit anhand von Beispielen und überschaubaren Übungsaufgaben zielgerichtet zu strukturieren sowie mit einfach handhabbaren Beispielen zu unterstützen, ist ein wesentliches Ziel dieses Buches.

Die Inhalte basieren auf der Vorlesung „Softwareentwicklung“, die ich seit 2003 für Studierende der Elektrotechnik an der Technischen Hochschule Mittelhessen in dieser Form anbiete. Neben vielen Beispielen enthält das Buch eine Fülle von Übungsaufgaben mit ausführlichen Lösungsvorschlägen. Wesentliche Teile der Lösungsvorschläge sind jeweils am Ende des zugehörigen Hauptkapitels abgedruckt. Zudem stehen die vollständigen Projekte für Visual Studio 2010 auf der Seite http://www.oop-fuer-ingenieure.de.vu/ zum Herunterladen zur Verfügung.

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

Ich danke allen an dieser Arbeit beteiligten Studenten und Mitarbeitern der Technischen Hochschule Mittelhessen, insbesondere Dipl. Ing. (FH) Johannes Friedrich für seinen engagierten Einsatz bei der Umsetzung der Inhalte in konkreten Lehrveranstaltungen. Ein besonderer Dank gebührt meinem Sohn Malte, der das gesamte Manuskript Korrektur gelesen und dank seines überragenden Sprachgefühls an entscheidenden Stellen in eine gute Form gebracht hat.

Gießen, im September 2014 *Uwe Probst*

URL der Internetseite mit den Applets zum Buch:

http://www.oop-fuer-ingenieure.de.vu/

Inhalt

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 mit Hilfe 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 mit Hilfe 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
http://www.oop-fuer-ingenieure.de.vu

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.

Index