

# Objektorientierte Programmierung für Dummies

Marcus Bäckmann

mitp-Verlag, ISBN 3-8266-2984-1

Nachtrag zu Kapitel 19

Die folgenden Teile erscheinen nicht im Buch, da sie aus Platzgründen gekürzt wurden – der Text würde eigentlich ab Seite 359 im Buch erscheinen... leider wurde die Sache ein wenig dick.

## 19 Mehr Container und Adapter

### In diesem Kapitel

- spielen Sie Herzblatt, indem Sie Pärchen mit Hilfe von `map<Key, T>` verheiraten

<Seite 351 – 359>

### 19.3 Stack

Ein ebenso klassischer Fall wie eine Queue als Datenstruktur ist der *Stack*, der *Stapel*. Im Prinzip ist der Stack ähnlich eingeschränkt wie eine Queue, nur dass man diesmal den Einhäng- und Aushängvorgang nur am gleichen Ende durchführen kann. Wie der Posteingangskorb, auf dem man die neuen Rechnungen immer oben auflegt – und herausnehmen darf man immer nur die oberste Rechnung. Auch das schreit wieder nach einer praxisgerechten Simulation.

```
#include <stack>
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;
class Bill
{
public:
    Bill(float moneymoney)
        : m_Moneymoney(moneymoney)
    {}
    float pay()
    {
        float moneymoney = m_Moneymoney;
        m_Moneymoney = 0.0f;
        return moneymoney;
    }
private:
```

```

float m_Moneymoney;
};
int main()
{
    stack<Bill> mybills;
    srand( (unsigned)time( NULL ) );
    for (int i = 0; i < 20; i++)
    {
        mybills.push(Bill(static_cast<float>(rand()) /
                        RAND_MAX * 100.0f));
    }
    cout << "Es sind " << mybills.size()
         << " Rechnungen zu zahlen\n";
    cout << "Wie viel Geld haben Sie? ";
    float account;
    cin >> account;
    while (account >= 0.0f)
    {
        account -= mybills.top().pay();
        mybills.pop();
        if (mybills.empty())
            break;
    }
    if (mybills.empty())
    {
        cout << "Herzlichen Glueckwunsch!" << endl;
    }
    else
    {

```

```

        cout << "Bleiben noch " << mybills.size()
            << " Rechnungen und das Konto ist bereits"
            << " in den Miesen!\n";
    }
    return 0;
}

```

**Listing 1:** *KAP20/STACK\_DEMO.CPP*

Wenn Sie einen ausreichend großen Geldbetrag eintippen, wird wohl tatsächlich die Erfolgsmeldung

Herzlichen Glueckwunsch!

am Bildschirm erscheinen. Falls Ihnen das Geld ausgeht, werden Sie wohl mit der Ausgabe

Bleiben noch 4 Rechnungen und das Konto ist bereits in den Miesen!

beglückt werden. Wie im richtigen Leben, diese Programme.

Wenn Sie die Programme `STACK_DEMO.CPP` und `QUEUE_DEMO.CPP` vergleichen, stellen Sie wahrscheinlich schon eine starke Ähnlichkeit in der Schnittstelle dieser beiden Containerklassen fest. Auch die Referenz der Memberfunktionen in Tabelle 19.3 verdeutlicht dies noch einmal.

Memberfunktion	Beschreibung
<code>stack</code>	erzeugt einen leeren Stack
<code>T&amp; top()</code>	gibt eine Referenz auf das letzte Objekt zurück
<code>bool empty()</code> <code>const</code>	liefert <code>true</code> , falls der Stack leer ist
<code>void pop()</code>	entfernt das oberste Objekt vom Stack
<code>void push(const</code> <code>T&amp; val)</code>	legt eine Kopie des Objekts <code>val</code> oben auf dem Stack ab
<code>size_type</code> <code>size() const</code>	liefert die Anzahl der gespeicherten Werte zurück

*Tabelle 19.3: Wichtige Memberfunktionen der Template-Klasse `stack<class T>`*

Weitere wissenswerte wunderliche Wesensmerkmale von `stack<T>` finden Sie in der folgenden Auflistung.

## DEFINE

- Ein *Stack (Stapel)* ist eine Datenstruktur, bei der Daten nur oben aufgelegt und oben entfernt werden können. Stacks sind LIFO-Speicher, je früher ein Element abgelegt wurde, desto später wird es auch entfernt. In der STL wird dies durch die Containerklasse `stack<class T>` realisiert, die sich im Namensraum `std` befindet und zu der der Header `<stack>` gehört.
- Aus historischen Gründen spricht man bei einer Queue vom vorderen und hinteren Ende, aber bei einem Stack von oben und unten. Wahrscheinlich wegen der gedanklichen Vorstellung, dass man einen Stapel Papier oder Akten vor sich hat. Deswegen gibt es für den Zugriff auf das erste Element hier die Memberfunktion `top` und nicht etwa `front`.
- Auch die Klasse `stack<T>` ist nur ein Adapter für andere Containerklassen. Wie bei `queue<T>` lässt sich die intern verwendete Containerklasse ändern, als default wird `deque<T>` verwendet.
- Verwechseln Sie bitte die Containerklasse `stack<T>` nicht mit dem Stack des Betriebssystems, auf dem lokale Variablen gespeichert werden. Beide haben das gleiche LIFO-Funktionsprinzip, da enden aber auch die Gemeinsamkeiten bereits.
- Die Klasse `Bill` kann über eine flache Kopie kopiert werden, so dass hier weder Copykonstruktor noch Zuweisungsoperator explizit definiert werden müssen.
- Auch die Klasse `stack<T>` besitzt keine Iteratoren.

## WARNING

- Gefährlich ist es bei einem leeren Stack, mit `stack<T>::top` auf ein Element zugreifen zu wollen. Prüfen Sie vorher mit `stack<T>::empty`, ob der Stack auch wirklich Elemente enthält.

## 19.4 Maps

Die Containerklasse `map` ist eine wirklich mächtige Datenklasse der STL, aber wer sich damit befasst, stößt in manchem Buch auf solche Sätze:

*Die Klasse `map` stellt einen Container bereit, in dem eindeutige Schlüssel auf Werte abgebildet werden. Ihre Template-Spezifikation sieht so aus:*

```
template<class Key, class T, class Pred = less<Key>,
class A = allocator<T> > class map
```

*Hierbei ist `Key` der Datentyp der Schlüssel, `T` ist der Datentyp der zu speichernden Werte und `Pred` ist eine Funktion, die zwei Schlüssel miteinander vergleicht.*

Äh, gut, das klingt doch hervorragend. Was wollte uns der Autor damit sagen? Dieser Abschnitt über die Containerklasse `map` wurde aus aktuellem Anlass noch zusätzlich zum Buch hinzugefügt, da mir in diesen Tagen ein Problem im C++-Forum aufgefallen war. Ein Frager hatte dort das Problem, dass er vom Betriebssystem Nachrichten mit einem Handle eines Fensters, also einem eindeutigen Kürzel, geschickt bekam, und er mit den Handles jeweils bestimmte Fenster-Objekte verbinden wollte. Er brauchte also eine Datenstruktur,

- ✓ die zwei verschiedene Typen miteinander verbindet
- ✓ bei der man nach einem Schlüssel (dem Handle) suchen kann und als Ergebnis das Objekt zurückerhält

Genau dies leistet die Klasse `map<Key, T>`. *Key* bezeichnet hierbei einen Schlüssel-Typ, nach dem man sucht. Das kann eine Postleitzahl sein – oder im obigen Problemfall das Handle. *T* ist der Datensatz, der mit dem Schlüssel verbunden ist, also zur Postleitzahl ein String mit dem Namen der Ortschaft, oder eben das Objekt des zum Handle gehörenden Fensters. Das folgende Programmbeispiel enthält einen Lösungsansatz für das Problem unseres Fragers, der Handles schon hat und nun nach Objekten sucht.

```
#include <map>
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;
class Window
{
public:
    Window(int id)
        : m_Id(id)
    {}
    int getID() const
    {
        return m_Id;
    }
private:
```

```

    int m_Id;
};
typedef unsigned HANDLE;
const HANDLE maxHandles = 10;
int main()
{
    map<HANDLE, Window> windowbase;
    for (HANDLE i = 1; i <= maxHandles; i++)
    {
        windowbase.insert(
            pair<HANDLE, Window>(i, rand()));
    }
    for (;;)
    {
        cout << "Geben Sie ein Handle ein:";
        HANDLE hnd;
        cin >> hnd;
        if (hnd == 0)
            break;
        map<HANDLE, Window>::iterator it
            = windowbase.find(hnd);
        if (it != windowbase.end())
        {
            cout << "Das zugehoerige Window enthaelt "
                << "die ID: "
                << it->second.getID() << endl;
        }
        else
        {

```

```

        cout << "nicht gefunden" << endl;
    }
}
return 0;
}

```

Eine kurze Erklärung zur Funktionsweise des Programms erhalten Sie im Abschnitt *Zurechtfinden mit der map*.

((Kasten Anfang))

BLOW

### ***Zurechtfinden mit der map***

Als zu verwaltende Klasse wird `window` definiert, diese Klasse kann einfach einen Wert speichern, die Informationen könnten aber auch wesentlich komplexer sein. Der Schlüssel, nach dem in der `map` gesucht wird, bekommt den Typ `HANDLE`, was nur ein Synonym zu einem `unsigned` ist. Daraus ergibt sich die folgende Definition des Container-Objekts:

```
map<HANDLE, Window> windowbase;
```

Eingehängt in den Container werden nun Paare aus `HANDLE` und einem Objekt der Klasse `Window`:

```

windowbase.insert (
    pair<HANDLE, Window>(i, rand()));

```

Dazu dient die Template-Klasse `pair`, die im Header `<utility>` definiert ist. Der erste Parameter `i` wird an `HANDLE` weitergereicht, der zweite Parameter – eine Zufallszahl – an den Konstruktor der Klasse `Window`. Mit `map<Key, T>::insert` wird dieses neue Objekt eingehängt.

Um nach einem solchen Paar wieder zu suchen, wird eine Variable `hnd` vom Typ `HANDLE` vom Benutzer eingegeben. Wenn man nun nach `hnd` sucht, möchte man das damit verbundene Objekt vom Typ `Window` wiederfinden. Gespeichert wird der Ort des Findens mit Hilfe eines Iterators.

```

map<HANDLE, Window>::iterator it
    = windowbase.find(hnd);

```

Die Funktion `find` liefert einen Iterator auf das gespeicherte `pair`-Objekt zurück, falls die Suche erfolgreich war. In diesem Fall kann man mit `*it` auf das Objekt zugreifen. Im Element `first` der Template-Klasse `pair` steht das

erste Objekt des Pärchens, in `second` das zweite Objekt. Also kann über `it->second` auf das `window`-Objekt zugegriffen werden:

```
<< it->second.getID() << endl;
```

Verläuft die Suche erfolglos, enthält der Iterator den Wert `map<Key, T>::end()`. Im Programmbeispiel ist das dann der Fall, wenn Sie Zahlen größer als 10 eingeben.

((Kasten Ende))

Interessant, was es so alles in der STL gibt, oder? Wenn Sie also mal wieder Daten speichern müssen oder wollen, die Sie über einen Schlüssel wiederfinden wollen, blättern Sie auf diese Seite und denken Sie darüber nach, das Problem mit Hilfe von `map<Key, T>` zu lösen.

DEFINE

- Die Containerklasse `map<class Key, class T>` ermöglicht die Speicherung von Wertepaaren aus einem Schlüsselobjekt der Klasse `Key` und einem Wertobjekt der Klasse `T`. Auf diesem Container ist eine Suche nach `key` möglich, um das zugehörige Objekt der Klasse `T` wieder zu ermitteln. Auch `map` befindet sich im Namensraum `std` und liegt im Header `<map>`.
- `map` erlaubt nur eindeutige Keys, es ist nicht möglich, einen vorhandenen Key mit einem anderen Objekt `T` ein zweites Mal im Container zu speichern. Wollen Sie so etwas tun, müssen Sie die Klasse `multimap` verwenden, die sich ebenfalls in `<map>` befindet.
- Will man komplexe Schlüssel miteinander vergleichen, muss man zusätzlich noch eine Funktion mitliefern, die den Vergleich zweier Schlüsselobjekte durchführen kann.
- Eine vollständige Referenz für `map` und `multimap` entnehmen Sie bitte der Hilfe Ihrer Entwicklungsumgebung oder greifen Sie online auf [http://www.dinkumware.com/htm\\_cpl/map.html](http://www.dinkumware.com/htm_cpl/map.html) oder <http://www.sgi.com/tech/stl/Map.html> zurück. Auch das Buch »C++ GE-PACKT« von Herbert Schildt, ebenfalls aus diesem tollen mitp-Verlag, enthält eine komplette Referenz dieser Klassen.
- Die Template-Klasse `pair<typename T1, typename T2>` aus dem Header `<utility>` speichert ein Paar aus zwei Objekten zweier Klassen `T1` und `T2`. Über `pair<T1, T2>::first` und `pair<T1, T2>::second` kann man auf das Objekt der Klasse `T1` bzw. der Klasse `T2` zugreifen.

TECHNICAL

- Statt der Zeile

```

windowbase.insert (
    pair<HANDLE, Window>(i, rand()));

```

kann man auch auf die Funktion `make_pair` zurückgreifen, in diesem Fall ändert sich die `insert`-Anweisung etwas

```

windowbase.insert (make_pair(i, rand()));

```

- Die Klasse `Window` kann über eine flache Kopie kopiert werden, so dass hier weder Copykonstruktor noch Zuweisungsoperator explizit definiert werden müssen. Beachten Sie, dass Objekte der Klasse `pair<HANDLE, Window>` kopiert werden, der Copykonstruktor von `pair` greift dazu auf die Copykonstruktoren von `HANDLE` und `Window` zurück.
- Die Template-Klasse `pair<T1, T2>` lässt sich auch benutzen, um mehr als einen Wert mit `return` zurückzugeben. Einige Leute vermissen solche Features – betrachten Sie das folgende kurze Beispiel, in dem eine Funktion eine Division durchführt. Falls durch 0 geteilt werden soll, liefert die Funktion einen zusätzlichen Fehlercode zurück.

```

#include <utility>
#include <iostream>
using namespace std;
pair<double, int> div(double a, double b)
{
    int errorcode = 0;
    double res = 0.0;
    if (b == 0.0)
        errorcode = 1;
    else
        res = a / b;
    return pair<double, int>(res, errorcode);
}
int main()
{
    pair<double, int> c = div(3.14, 0.0);
    if (c.second == 0)

```

```
    cout << c.first << endl;

    return c.second;
}
```

Im dargestellten Fall ist sogar das in *Übergabe von Objekten* erklärte Verfahren der inplace construction möglich, es werden also keine unnötigen Konstruktoraufrufe durchgeführt. Sie finden das Programm auch als KAP20/DOUBLERETURN.CPP im Download.

COMPILER

- Falls Sie Visual C++ zur Kompilierung der Programme verwenden, fügen Sie bitte gleich am Anfang des Programms die folgende Zeile ein

```
#pragma warning(disable : 4786)
```

Durch die vielen Templates und darin eingesetzten Typen werden hier die Namen der Klassen länger als 255 Zeichen und Visual C++ erzeugt deswegen jedes Mal eine sehr nervige Warnung, die mit dem #pragma abgeschaltet werden kann.