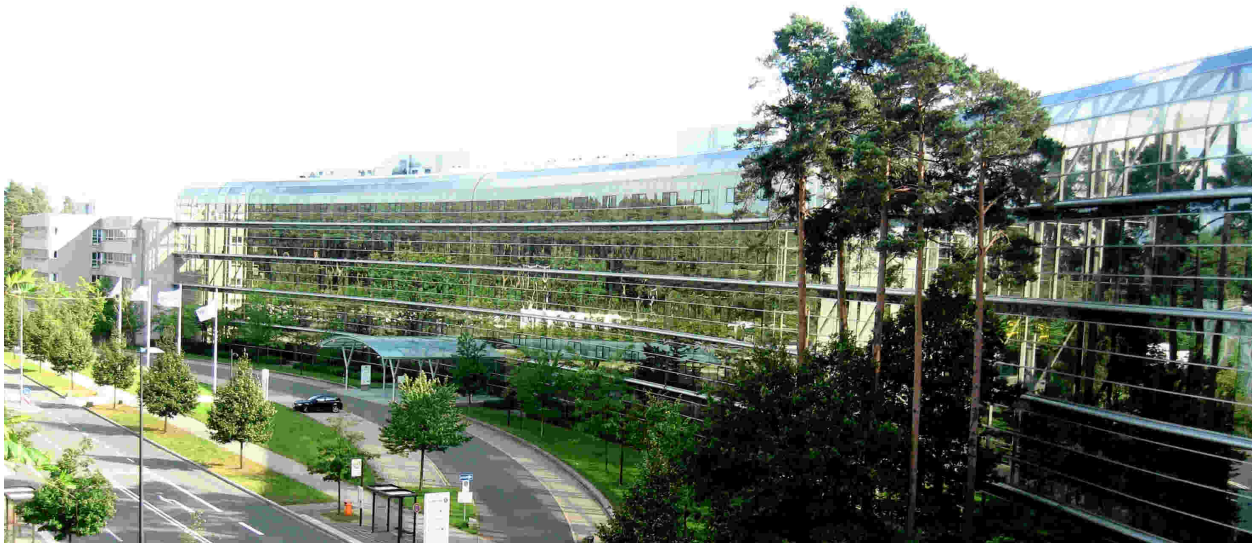
NewElements IT-Schulungen.com

C++/OOP - Seminar

Micro-Epsilon Messtechnik

Dr.sc.nat. Michael J.M. Wagner, New Elements*

Revision 1.20



*michael@wagnertech.de

Inhaltsverzeichnis

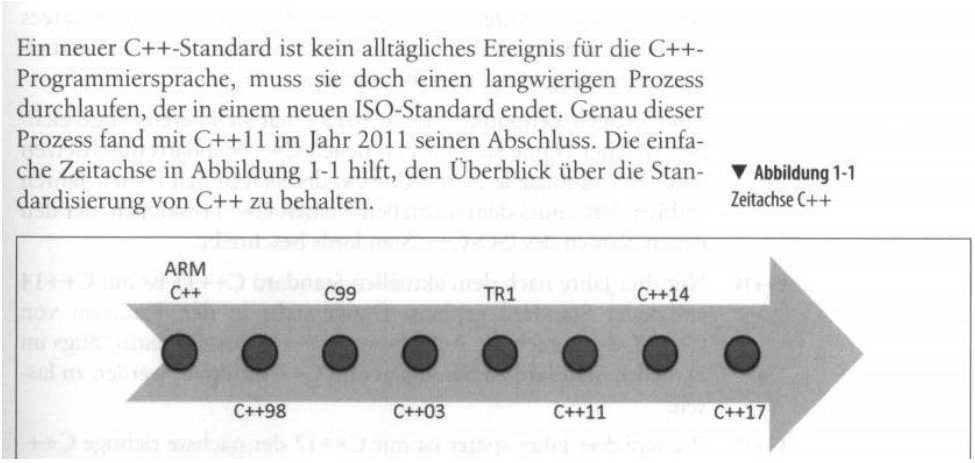


Abbildung 1: C++-Geschichte[2]

1 Einstieg in C++[1]

Als Urväter von C++ gelten *Smalltalk*, die erste objektorientierte Programmiersprache und C, *die* Spache für die Systemprogrammierung. C++ ist daher eine objektorietierte Sprache, die auch für die Systemprogrammierung geeignet ist.

Die Geschichte von C++ startet in den späten 70-ern. Mit dem Hype der Objektorietierung, der nach und nach aufkam, hasst alles „objektorieniert" zu sein. Der erste Titel war daher 1979 „C with classes". 1983 wurde der Template-Mechanismus hinzugefügt und das Ganze hieß nun C++.

Während des Hype der Objektorietierung wurde die Ansicht vertreten, dass allein die Einführung der Objektorientierung zu wiederverwendbarer und wartbarer Software führt. Neue nützlich Klassen werden entwickelt und gemeinsam genutzt. Aber was passierte: Nützliche Klassen wurden für dasselbe Problem immer wieder auf's Neue entwickelt. In den 90-ern existierte damit eine Vielzahl von Klassen beispielsweise zur Behandlung von Zeichenketten. Jeder Compiler-Hersteller lieferte seine eigene *string*-Klasse. Und diese Klassen waren nicht kompatibel. Daher wurde ein Standardisierungsprozess gestartet. Die Schritte der Standardisierung zeigt Abbildung 1.

- *Annotated Reference Manual*: Entwürfe der Standardisierung
- 1998: ANSI-C++

  Dieser Standard enthält die wichtigen Klassenbibliotheken. Die meisten sind Template-Klassen. Daher wird dieser Standard oft STL (*standard template library*) genannt.
- 2003: Verbesserungen zu C++98

  Wichtigste Neuerung: `auto_ptr`, eine erste Klasse zur sicheren Freispeicherverwaltung.

[1] Wolf: Kap. 1
[2] Grimm: S. 3

- 2011: C++11

  2011 gab es keinen Hype der Objektorietierung mehr. *Functional programming* hielt nun Einzug in die Sprache. Was bedeutet dies? *Functional programming* vermeidet Schleifen. Wenn ein Algorithmus auf eine Menge Daten angewendet werden soll, wird der Algorithmus definiert und dem Compiler mitgeteilt, dass dieser Algorithmus auf jene Daten anzuwenden sei. Dieser Ansatz ermöglicht dem Compiler eine implizite Parallelisierung. Für eine einfache Formulierung von Algorithmen wurden die *Lambda-Funktionen* eingeführt. Eine Lambda-Funktion ist eine *ad hoc*-Definition einer Funktion.

- 2014: C++14: Verbesserungen zu C++11

  Die Lambda-Funktionen erhalten neue Eigenschaften. In Kombination mit dem `auto`-Schlüsselwort können nun auch generische Lambda-Funktionen geschrieben werden.

- 2017: C++17: Verbesserungen zu C++11

  Standardisierung der Parallelisierung

- 2020: C++20: Konzepte, Koroutinen, Attribute

Ziele von C++ 11[3]

- Für den Einsteiger: einfacher zu lernen
- Für den Profi: eine noch bessere Programmiersprache für die Systemprogrammierung
- Multiparadigmen-Progammiersprache
  - prozedural, strukturiert (C)
  - objektorientiert, generisch (C++98)
  - funktional (C++11): Vermeidung von Schleifen, Zuweisungen, ermöglicht dem Compiler eine Parallelisierung

Das erste C++-Programm[4]

Ein erstes einfaches Programm ist in Abbildung 2 abgebildet. Bevor es an die Praxis geht, sollen folgende Sprachelemente besprochen werden:

- Blöcke
- Ein- Ausgabeströme

  In der Sprache C++ gibt es keine Funktionen zur Ein- oder Ausgabe, die Bestandteile der Sprache selbst sind. Stattdessen wird die einfache Ein- und Ausgabe über die (objektorientierte) Streambibliothek `<iostream>` definiert:

  - `std::cout`: Standard-Ausgabestream
  - `std::cerr`: Standard-Fehlerausgabestream

[3] Grimm: S. 7
[4] Wolf: Kap. 2

```
#include <iostream>
/*
 * Hauptprogramm
 */
int main() {
  std::cout << "Mein erstes Programm" << std::endl;
  int i;
  // hier wird i durch eine Tastatureingabe belegt.
  std::cin >> i;
  return 0;
}
```

Abbildung 2: Das erste Programm in C++

  - `std::cin`: Standard-Eingabestream
- Bezeichner: Variablen, Funktionen, Klassen
- Literale
- Kommentare

Aufgabe:

Erstellen Sie in Ihrer Entwicklungsumgebung ein neues Projekt und bringen Sie ein einfaches „Hallo Welt"-Programm zum Laufen.

2 Sprachelemente

2.1 Spracherweiterungen gegenüber C

C++ baut auf C auf. Daher können die C-Sprachelemente auch in C++ verwendet werden. In der Kernsprache gibt es seit C++98 folgende Erweiterungen:

- Streams für die Ein- Ausgabe (Kap. 3)
- Zeilenkommentare: `//`
- Referenzen (Kap. 2.3)
- Default-Argumente (Kap. 2.4)
- Funktionen überladen (Kap. 2.4)
- Diverse Sprachelemente für die Objektorientierung (Kap. 5)
- Namensräume (Kap. 2.5)
- Operatoren überladen (Kap. 5.2)

- Templates (Kap. 6)
- Exceptions (Kap. 4)

Mit C++11 wurde die Kernsprache um folgende Elemente erweitert:

- Vereinheitlichte Initialisierung
- Unicode-Unterstützung
- Neue Bedeutung des Schlüsselworts `auto`
- Range-for: `for (elem : collection)`
- `enum class`
- Typaliase mit `using`
- Lambdafunktionen
- Move-Semantik
- Variadic Templates
- `constexpr`

Für die Definition von Konstanten stehen nun drei Möglichkeiten zur Verfügung:

- `#define`-Direktive: Aus C bekannt, Ersetzen zur Compilezeit, nicht typsicher
- `const`: C++98, „echte" Variable, typsicher, kann ausgehebelt werden
- `constexpr`: C++11, Ersetzen zur Compilezeit, typsicher

2.2 Arrays und Zeichenketten[5]

In C ist die Handhabung von Arrays im Allgemeinen und die Handhabung von Zeichenketten als Array von Buchstaben alles andere als komfortabel. Mit C++98 kamen mit der Standardlibrary Klassen, die die Verwendung von Arrays und Zeichenketten erheblich erleichtern:

- Arrays: `std::vector<TYP>` [`006/listing001.cpp`, W:121]

  Der Elementzugriff über eckige Klammern erfolgt dabei ohne Bereichsüberprüfung. Will man eine solche, kann mit `my_array.at(<position>)` zugegriffen werden.

- Zeichenkette: `std::string` [`006/listing005.cpp`, W:132]

Für Zeichenketten gibt es in C++ nur wenig Funktionen. Grundsätzlich müssen bei Funktionsaufrufen zwei Notationen unterschieden werden:

- Prozedurale Notation: Unsere Zeichenkette ist ein Parameter eines Funktionsaufrufs: `f(string_var)`
- Objektorientierte Notation: Der Name der Zeichenkette wird mit einem Punkt *vor* den Funktionsaufruf gesetzt: `string_var.empty()` (liefert einen `bool`-Wert, ob eine Zeichenkette leer ist)

[5] Wolf: Kap. 6

Da die `std::string`-Implementiertung mit C++98 kam und zu dieser Zeit die Objektorientierung hoch im Kurs war, sind praktisch alle Funktionen für Zeichenketten in objektorientierter Notation. Eigenschaften von Zeichenketten[6]

Aufgabe:

Mit den nächsten Übungen soll Schritt für Schritt eine Bücherverwaltung für eine Bücherei erstellt werden. Führen Sie folgende Schritte aus:

- Legen Sie ein neues Projekt *Bucherei* an.
- Wandeln Sie in `Bucherei.cpp` das Beispiel `006/listing005.cpp` so ab, dass Sie vom Anwender 6 Werte (*Signatur, Autor, Titel, Typ, Seitenzahl, Spieldauer*) abfragen und diese in 6 Variablen speichern. Dabei seien *Signatur, Autor, Titel* Zeichenketten, *Typ* ein einzener Buchstabe, *Seitenzahl* und *Spieldauer* ganzzahlig.

  Anmerkung1: In C++ werden Variablennamen von einfachen Variablen üblicherweise in kleinen Buchstaben geschrieben. Die Namen können einen Unterstrich enthalten.

  Anmerkung2: Von der Konsole lesen wir grundsätzlich über `getline()`. Zur Umwandlung in einen `int` verwenden wir `std::stoi()`, zur Umwandlung in `char` nehmen wir den ersten Buchstaben der Zeichenkette.
- Eine Prüfung der Eingabe erfolgt nicht, wir gehen von einem gutmütigen Benutzer aus.
- Geben Sie die Variablen schön formatiert wieder aus.

2.3 Referenzen und Zeiger[7]

Zeiger und Referenzen sind Variablen, die auf eine andere Variable verweisen. Sie unterscheiden sich aber stark in der Syntax und ein wenig in der Verwendung. Der signifikateste Unterschied ist, dass ein Zeiger auch auf *nichts* (`NULL - nullptr`) zeigen darf, eine Referenz hingegen nie.

Beim Zugriff auf den Wert muss ein Zeiger erst dereferenziert werden (`* ->`), eine Referenz wird wie die Variable selbst verwendet. Eine Referenz lässt sich daher auch als Aliasname verstehen.

Zeiger: [`007/listing006.cpp`, W:147]

Referenzen: [`007/listing001.cpp`, W:138f]

2.4 Funktionen[8]

Parameterübergabe

Funktionen können Parameter haben. Per default erfolgt dabei ein *call by value*-Aufruf, d.h. die aufgerufene Funktion bekommt eine Kopie des Wertes. Änderungen in der gerufenen Funktion

[6] http://www.cplusplus.com/reference/string/string/
[7] Wolf: Kap. 7
[8] Wolf: Kap. 8

haben keine Auswirkung in der rufenden Funktion. [008/listing003.cpp, W:154f]

Wird der gerufenen Funktion eine Referenz auf eine Variable der rufenden Funktion übergeben, hat dies zwei Konsequenzen:

- Die Variable wird nicht in die gerufene Funktion kopiert.
- Änderungen in der gerufenen Funktion haben eine Auswirkung auf die rufende Funktion.

Funktionsdefinition mit Referenz[9]:

```
void addieren(int& val1, int val2) {
  // Diese Funktion addiert val2 zu val1
  val1 += val2;
}
```

Betrachtet man den ersten Punkt als Vorteil, da die übergebene Variable sehr groß ist, will man aber keine Rückwirkung in die rufende Funktion haben, so bietet sich die Verwendung einer konstanten Referenz an.

Funktionsdefinition mit konstanter Referenz[10]:

```
void addieren(const int& val1, const int& val2) {
  // Diese Funktion verwendet Referenzen auf die Variablen
  // der rufenden Funktion, darf sie aber nicht ver"andern
  std::cout << "Die Summe lautet: " << val1+val2 << '\n';
}
```

Zusammenfassung:

call by value „ohne &"

- Daten werden in das Unterprogramm kopiert
- keine Rückwirkung auf das Hauptprogramm
- Beim Aufruf auch mit Konstanten möglich

call by reference „mit &"

- Unterprogramm greift direkt auf die Variablen des Hauptprogramms zu
- Rückwirkung auf das Hauptprogramm
- Kein Aufruf mit Konstanten möglich

Anmerkung: Der Trend geht in eine neue Richtung: Eingabeparameter stehen in der Paramterleiste, Rückgabewerte werden als Tupel zurückgegeben. Das sieht dann folgendermaßen aus:

```
tuple<int,string> my_func(string s, int i) {
  string s_neu = s + "Welt";
  int i_neu = i + 5;
  return make_tuple(i_neu, s_neu);
}
```

[9] s. auch Wolf: Kap. 8.2.1
[10] s. auch Wolf: Kap. 8.4

```
int n = 5;
string str;
tie(str,n) = my_func("Hallo", n);
```

Funktionen mit Default-Argument: `void addieren(int val1 = 12, ...` [008/listing004.cpp, W:156f.]

Funktionen mit Rückgabewert: [008/listing005.cpp, W:159]

Aufgabe:

Zerlegen Sie das bisherige Bibliotheksbeispiel in Funktionen und verteilen Sie die Funktionen auf Dateien:

- Datei `InOut.cpp/.h` mit den Funktionen

  ```
  void readMedium(std::string& signatur, std::string& autor,
    std::string& titel, char& typ, int& seitenzahl, int& spieldauer);
  void writeMedium(const std::string& signatur,
    const std::string& autor, const std::string& titel,
    char typ, int seitenzahl, int spieldauer);
  ```

- Datei mit dem Hauptprogramm:

  Hier werden als erstes die sechs Variablen definiert, dann durch den Aufruf von `readMedium()` belegt und mit `writeMedium()` ausgegeben.

Funktionen überladen

In C++ können Funktionen mit demselben Namen, aber unterschielichen Parametern verwendet werden:

```
int rechnen ( int ivar );
int rechnen ( int ivar1, int ivar2 );
double rechnen ( double dvar ) ;
double rechnen ( double dvar1, double dvar2 );
```

Findet der Compiler keine direkt passende Funktion, so versucht er über Typumwandlungen eine zu finden.

Aufgabe[11]:

Schreiben Sie ein Programm, das die Fläche eines Rechtecks berechnet. Die Fläche ermitteln Sie mit Länge x Breite.

- Schreiben Sie die Funktion zur Flächenberechnung in drei Varianten:
  - mit zwei Integer-Eingabe-Parameter und einem Integer-Rückgabewert
  - mit zwei Float-Eingabe-Parameter und einem Float-Rückgabewert
  - mit zwei Float-Eingabe-Parameter und einem Float-Ausgabeparameter

Die drei Funktionen sollen denselben Namen haben.

- Verwenden Sie alle Funktionen im Hauptprogramm.

2.5 Namensräume[12]

Verfolgt man bei der Programmierung das prozedurale Paradigma, so ist es notwendig Namensräume zu schaffen, um die Verwendung gleichnamiger Funktionen in verschiedenen Kontexten zu ermöglichen. In den 90'er Jahren wurde dabei ausschließlich auf Sprachkonstrukte der Objektorientierung (s. Kap 5) gesetzt. Einfache prozudurale Funktionen werden in Klassendefinitionen als „statische Elemente" (`static`) definiert:

```
class MyClass {
public:
  static int foo() {
    return 47;
  }
}

int main() {
  int result = MyClass::foo();
}
```

Später kam das Namensraum-Konzept hinzu.

Die Verwendung von Namensräumen erfordert folgende Maßnahmen:

- Deklaration von Funktionen/Klassen innerhalb eines `namespace`-Blocks
- Definition von Funktionen innerhalb eines `namespace`-Blocks oder mit expliziter Angabe des Namensraums:
- Bei der Verwendung muss der Namensraum mit angegeben werden: [`009/listing001.cpp`, W:188u,189,191f]
- Ein Namensraum kann komplett importiert werden. Dies sollte man aber nur in Quelldateien machen, da die Verwendung in Headerdateien Nebenwirkungen haben könnte:
  `using namespace VIP_Bereich`
- Alternativ können auch nur einzelne Elemente eines Namensraums in den aktuellen Kontext importiert werden:
  `using VIP_Bereich::funktion;`

Aufgabe:

Legen Sie für die Funktionen `readMedium()` und `writeMedium()` Namensräume an:

[11] Wolf: S. 178ff.
[12] Wolf: Kap. 9.2

- In `InOut.h` werden die Deklarationen in eine Namensraumdefinition (`namespace InOut {`) gefasst.
- In `InOut.cpp` muss bei der Implementierung der Funktionen den Funktionsnamen ein `InOut::` vorangestellt werden.
- Auch beim Aufruf im Hauptptogramm muss ein `InOut::` vorangestellt werden.
- Importieren die im Hauptprogramm den Namensraum `std` komplett und fügen Sie am Ende des Programms eine Ausgabe hinzu.

3 Ein- und Ausgabe von Dateien[13]

Zum Lesen und Schreiben von Dateien stehen die Klassen `std::ifstream` und `std::ofstream` zur Verfügung.

Folgende Dateioperationen werden unterstützt:

- Öffnen von Dateien

```
#include <fstream>
std::ofstream file01("testdatei001.dat");
std::ifstream file02("testdatei002.dat");
if ( ! file02 ) {
  std::cerr << "Datei existiert nicht!" << std::endl;
}
std::ifstream file03;
file03.open("testdatei003.dat");
if (file03.fail()) {
  std::cerr << "Datei existiert nicht!" << std::endl;
}
// Datei testdatei004.dat wird am Ende beschrieben
std::ofstream file04("testdatei004.dat", std::ios::app);
```

- Schließen von Dateien: `data01.close();`

  Im Allgemeinen ist es nicht nötig Dateien zu schließen, da dies mit dem Ende des Blocks automatisch geschieht.

- Zeilenweises Lesen und Schreiben

```
#include <fstream>
using namespace std;
ifstream rStream("datei.txt");
ofstream wStream("out.txt");
string line;
while (getline(rStream, line)) {
  wStream << line << endl;
}
```

[13] Wolf: Kap. 17

- Für das blockweise Lesen und Schreiben stehen die Streammethoden `read` und `write` zur Verfügung.[14]

Aufgabe:

Ergänzen Sie die Bibliotheksanwendung:

- Legen Sie die Dateien `Medienverwaltung.h` und `Medienverwaltung.cpp` an.
- Schreiben Sie eine Funktion `addMedium` mit folgender Signatur:

  ```
  int addMedium(const std::string& signatur, const std::string& autor,
    const std::string& titel, char typ, int seitenzahl, int spieldauer)
  ```

  Die Deklaration kommt in die Header- die Definition in die Quelldatei.
- Vergessen Sie nicht, die Headerdateien in den Quelldateien zu inkludieren.
- `addMedium` soll die Daten an die Datei `medien.csv` kommasepariert anhängen.
- Rufen Sie die Prozedur aus dem Hauptprogramm mit den Werten aus `readMedium()` auf.

Liest man eine Datei zeilenweise ein, muss die gelesene Zeile in ihre Bestandteile zerlegt werden und diese Bestandteile dann in Variablen des richtigen Typs abgelegt werden. In C++ gibt es bis heute keine wirklich komfortablen Mittel. Mit den Dateien `util.h/.cpp` stehen diese Hilfsfunktionen zur Verfügung:

- `vector<string> tokenize(const string& line, char c);`
  wandelt `line` in einen `vector<string>` um, der die am Trennzeichen `c` getrennten Bestandteile von `line` enthält.
- `char toChar(const string& str);`
  wandelt eine Zeichenkette in einen Charakter, indem das erste Zeichen der Zeichenkette als `char` zurückgegeben wird.
- `int toInt(const string& str);`
  wandelt eine Zeichenkette in eine Ganzzahl.

Aufgabe:

Nehmen die Dateien `util.h/cpp` in Ihr Projekt auf und übersetzen Sie diese.

[14] Wolf: Kap. 17.3.6

4 Fehlerbehandlung[15]

Zur Fehlerbehandlung gibt es verschiedene Ansätze:

- Rückgabewerte vom Typ `int`
- Rückgabewerte eines speziellen Fehlertyps
- Exceptions
- Eigene Routine zur Ermittlung des Fehlerstatus

Empfehlung:

- Zu erwartende (oft fachliche) Fehler werden auf Rückgabewerte abgebildet. In diesem Fall kann direkt an der Aufrufstelle auf den Fehler reagiert werden.
- Unerwartete Fehler (oft technische Fehler, Logikfehler) werden über Exceptions behandelt. Der Programmablauf wird abgebrochen und ein einem Fehlerhandler, der üblicherweise „oben" in der Aufrufhierarchie sitzt, fortgesetzt.

Grundsätzlich kann in C++ „alles" als Ausnahme geworfen werden. Mit C++98 wurden Standardausnahmen definiert, deren Verwendung empfohlen sei.

Anwendung von Standardausnahmen: [`listings/016/exc005/main.cpp`, W:403f]

Werden in einem Projekt weitere Ausnahmetypen benötigt, so sollen die selbstdefinierten Ausnahmen von den Standardausnahmen abgeleitet werden.

Sollen nach und nach alle Ausnahmen gefangen werden, müssen zuerst die speziellen, zuletzt die allgemeinen gefangen werden:

```
#include <stdexcept>
try {
  // hier kommen Aufrufe, aus denen m"oglicherweise Ausnahmen
  // fliegen k"onnen
}
catch (std::runtime_error& e) {
  cout << "Es ist ein Runtime Error aufgetreten: " << e.what() << std::endl;
}
catch (std::exception& e) {
  cout << "Es trat folgendes Problem auf: " << e.what() << std::endl;
}
catch (...) {
  cout << "Es trat eine unbekannte Ausnahme auf." << std::endl;
}
```

Aufgabe:

Schreiben Sie eine Funktion

`bool isSignatureInFile(const std::string& signatur)`, die

[15] Wolf: Kap. 16

- prüft, ob die Datei `medien.csv` zum Lesen geöffnet werden konnte. Falls nein, darf eine Datei mit der Signatur angelegt werden, d.h. die Bearbeitung wird mit `return false;` beendet.
- Lesen Sie die Datei Zeile für Zeile.
- Prüfen Sie auch, ob jede Zeile 6 Tokens hat (`tokens.size()`). Falls nein, werfen Sie eine Exception.
- Trennen Sie die Signatur ab und vergleichen Sie diese mit der übergebenen.
- Falls die Signatur schon vorhanden ist, geben Sie `true` zurück.
- Wird die Signatur bis zum Dateiende nicht gefunden, wird `false` zurückgegeben.

Ergänzen Sie die Funktion `addMedium` um Rückgabewerte. Legen Sie dazu in der Headerdatei passende Integerkonstanten an:

```
constexpr int RC_OK = 0;
constexpr int RC_DUPLIKAT = 1;
```

- Rufen Sie die Funktion `isSignatureInFile` auf.
- Reagieren Sie bei einem Duplikatsfehler mit der Rückabe des entsprechenden Fehlercodes.
- Werten Sie im Hauptprogramm den Rückgabewert aus und fangen Sie die Ausnahmen auf.

5 Objektorientierung

Der rein prozedurale Ansatz kommt in großen Projekten an seine Grenzen. Ein Problem ist, dass es immer wieder Namenskonflikte im globalen Namensraum gibt, ein weiteres, dass die Verantwortlichkeit für den Inhalt von Datenstrukturen unklar ist.

Die Objektorientierung galt in den 90er-Jahren als Allheilmittel. Daher wurden Klassendefinitionen als Behälter für Prozeduren missbraucht. Es zeigte sich aber bald, dass auch Klassennamen zu Namenskonflikten führen können. Daher wurden Namensräume (*namespaces*) eingeführt (s. Abschnitt 2.5. In C++ besann man sich wieder darauf, dass prozedurales Denken nicht per se unanständig ist. Gerade mit C++11 kam wieder viel funktionales Denken in die Sprache (zurück).

Folgene Erfahrungen in der Softwareentwicklung haben zur Idee der Objektorientierung geführt:

- Strukturen sind eine sehr nützliche Sache, um Daten, die logisch zusammen gehören, zusammen zu verwalten.
- Wird eine Struktur im Speicher angelegt, wurde es in großen Programmwerken schnell unübersichtlich, wer diese Struktur für welchen Zweck gebraucht und wer Änderungen daran vornimmt.

- Unklar war oft, ab welchem Zeitpunkt welche Bestandteile einen gültigen Wert besitzen.

Vor diesem Hintergrund kam die Idee auf, die Zugriffe auf Strukturen (lesend, wie schreibend) zu kontrollieren. Der allgemeine Zugriff auf die Datenstruktur wurde also verboten, stattdessen wurden Funktionen geschaffen, über die auf die Daten zugegriffen werden konnte. Eine Datenstruktur mit den dazugeförigen Zugriffsfunktionen nennt sich *Klasse*.

Wie in C lassen sich in C++ Datenstrukturen definieren. Die Instanzierung bedeutet dann, dieser Struktur einen konkreten Speicherbereich zuzuordnen. Wie auch in C muss dabei beachtet werden, wo die Daten instanziert werden: Lademodul, Stapel oder Freispeicher. In letzerem Fall ist der Entwickler auch für dessen Freigabe verantwortlich. Nur eine instanzierte Struktur (=Objektinstanz) kann auch verwendet werden.

Bei einer Klasse sind im Gegensatz zu einer Struktur die interen Datenelemente von außen nicht zugreifbar (`private`). Alle Zugriffe erfolgen prozedural über öffentliche (`public`) *Methoden*.

Auf der anderen Seite dienen Klassen oft als Container, um Funktionen, die in logischem Zusammenhang stehen, in einer gemeinsamen Einheit zusammenzufassen. Sie bilden ein „Funktionsmodul". Funktionen einer Klasse, die nicht auf der „internen" Datenstruktur arbeiten, nennen sich *statisch* (`static`). Zur Verwendung von statischen Funktionen muss zuvor keine Objektinstanz angelegt werden.

Real existierende Klasse befinden sich irgenwo im Spektrum zwischen „Funktionsmodul" und „Datenverwaltungsmodul".

5.1 Klassen[16]

Klassendefinition:[17]

```
class Klassenname {
// Auf Elemente kann nur innerhalb einer Klasse
// zugegriffen werden
private:
typ daten1;
typ daten2;
...
// Zugriff von aussen auf die Elemente m"oglich
public:
typ funktionsname1( parameter );
typ funktionsname2( parameter );
...
};
```

Ein Beispiel findet sich in `listings/011/automat1/automat.h`[18]

Die Definition der Klassenmethoden erfolgt in der zugehörigen `cpp`-Datei:
`listings/011/automat1/automat.cpp`[19]

[16] Wolf: Kap. 11
[17] Wolf: S. 238.
[18] Wolf: S. 239.
[19] Wolf: S. 240.

Greifen Klassenmethoden auf Datenelemente der Instanz zu, kann dies mit einem `this->` verdeutlicht werden:

```
string Automat::get_standort() const {
   return this->standort;
}
void Automat::set_standort(const string& standort) {
   this->standort = standort;
}
```

Greifen Klassenmethoden nur lesend auf die Datenelemente der Instanz zu, kann dies mit `const` versichert werden.

Instanzierung auf dem Stapel:

```
Klassenname Objekt;
Klassenname Objekt{}; // C++11

// Beispiel f"ur Klasse Automat:
Automat device01;
Automat device01, device02;

Automat device01{}; // C++11
Automat device01{}, device02{}; // C++11
```

Aufruf von Klassenmethoden (Punktoperator):

```
// Objekt der Klasse "Automat" erzeugen
Automat device{};
// Daten des Objektes "andern
device.set_geld(20000);
device.set_standort("Augsburg, Fuggerweg 345");
// Inhalt des Objektes ausgeben
device.print();
```

Die Instanzierung im Freispeicher erfolgt mit `new`:

```
Automat* aptr = new Automat(); // Instanzierung
aptr->set_geld(20000);         // Verwendung
delete aptr;                   // Speicherfreigabe
```

Da das `delete` oft nicht korrekt implementiert wird, gibt es seit 2003 den `auto_ptr`, der mit C++11 durch den `unique_ptr` ersetzt wurde.

```
#include <memory> // f"ur std::unique_ptr
...
// Speicher f"ur ein neues Objekt anfordern
std::unique_ptr<Automat> device_ptr(new Automat{});
std::unique_ptr<Automat> device_ptr = make_unique<Automat>(); // Alternative
auto device_ptr = make_unique<Automat>(); // Alternative mit make_unique / auto
device_ptr->set_standort("Mainz, Hauptstrasse 1");
device_ptr->print();
```

Weitere Funktionen von `unique_ptr`:

- `*device_ptr`: Das dereferenzierte Objekt
- `device_ptr.get()`: Der Zeiger selbst. Die Verantwortung bleibt aber beim `device_ptr`.

- `device_ptr.release()`: Der Zeiger selbst. Die Verantwortung geht auf den Empfänger über, `device_ptr` selbst wird leer.
- `device_ptr.reset(p)`: `device_ptr` übernimmt den neuen Zeiger `p`. Falls bereits in `device_ptr` ein Zeiger gespeichert war, wird das zugehörige Objekt gelöscht.

Konstruktoren und Destruktoren

Konstruktoren sind Funktionen, die bei der Instanzierung einer Klasse, unabhängig davon, wo sie instanziert wird, ablaufen. Sie dienen meist der Initialisierung. Konstruktoren können Parameter aufnehmen. Standardmäßig steht der Default-Konstuktor (ohne Parameter) zur Verfügung. Dieser wird aber ausgeblendet, sobald ein spezieller Konstruktor definiert wird. In C++11 lässt sich dieser mit `MyClass() = default;` wieder einblenden.

Deklaration von Konstruktoren: [`011/automat2/automat.h`, W:247]

Definition von Konstruktoren:

```
Klassenname::Klassenname( )
: data1{wert}, data2{wert}, dataN{wert} {
  // Konstruktionsk"orper mit Anweisungen
}
```

Sollen, wie im gezeigten Beispiel, im Konstruktor keine expliziten Prüfungen durchgeführt werden, wenn der Funktionskörper also leer ist, dann empfiehlt es sich, den Konstruktor *inline* in der Headerdatei zu definieren. Dazu lässt sich seit C++11 auch die Konstruktor-Delegation verwedenden [`011/automat2/delegation/automat.h`, W:253].

Es gibt Klassenelemente mit speziellen Signaturen, die der Compiler in bestimmten Situationen heranzieht. Grundsätzlich gilt hier ein „alles oder nichts". Ist spezieller Code zum Kopieren oder Löschen eines Objekts nötig, müssen alle drei (mit C++11 fünf) Elemente berücksichtigt werden:

- Kopierkonstruktor: `Class(const Class&)`
- Zuweisungsoperator: `Class& operator=(const Class&)`
- Destruktor: `~Class()`
- Move-Konstruktor (C++11): `Class(Class&&)`
- Move-Zuweisung (C++11): `Class& operator=(Class&&)`

In modernem C++-Code, der die Möglichkeiten der Standardlibrary nützt, sollte möglichst auf die Verwendung dieser Sprachelemente verzichtet werden (*rule of zero*[20]):

- Verwenden Sie für größere Datenmengen fertige Container der Standardbibliothek.
- Benutzen Sie die neuen klugen Zeiger.

Ergänzen Sie die Bibliotheksanwendung

[20] Wolf: S. 291f.

- Erstellen Sie eine Klasse `Benutzer` mit Nutzernummer, Name, Vorname, Ort als *members* mit Defaultwerten. In den objektorientierten Sprachen werden Klassen üblicherweise in gleichnamige Dateien (`Benutzer.cpp/.h`) gelegt.
- Ergänzen Sie die Klasse `Benutzer` um zwei Konstruktoren, der die einzelnen Bestandteile als Parameter übernehmen, einmal mit Nutzernummer, einmal ohne. In letzterem Fall wird diese mit Null belegt.
- Schreiben Sie zu jedem Datenelement einen *getter* (z.B. `string getName() const;`, hier kann die Entwicklungsumgebung helfen!)
- Erstellen Sie das Modul *Nutzerverwaltung* mit der Funktion `int anlegen(const Benutzer& nutzer)`, die erstmal nichts weiter als eine Ausgabe auf der Konsole erzeugt.
- Ergänzen Sie das Hauptprogramm um die Eingabe von Nutzerdaten, instanzieren Sie damit einen Benutzer und rufen Sie diese Funktion.

5.2 Operatoren[21]

In C++ können die bestehenden Operatoren „überladen" werden, das heißt, mit „Sinn" für die betreffende Objektklasse versehen werden. Interessant ist hierbei die Definition des Streamoperators, damit bei Ausgaben ein Objekt „schön" dargestellt wird.

Für die Überladung von Operatoren gibt es zwei Syntax-Möglichkeiten:

- Definition als Methode einer Klasse. In diesem Fall ist die Klasse selbst implizit der Typ des ersten Operanden.

  Vorteil: Bei der Implementierung des Operators kann auf private Daten des Klasse zugegriffen werden [`013/dint01/dint.h`, W:321f].

- Definition als globale Funktion.

  Vorteil: Die Typen der Operanden unterliegen keiner Einschränkung.

Muss bei der Implementierung eines Operators als globale Funktion dennoch auf private Elemente eines der Operanden zugegriffen werden, so eignet sich das `friend`-Konzept. In einer Klassendefinition kann eine globale Funktion als Freund bezeichnet werden. Diese darf dann auch auf private Daten zugreifen [`013/dint02/dint.cpp`, W:326f].

Überladen des Ein-/Ausgabeoperators: [`013/dint06/dint.h/.cpp`, W:336f,337ff]

Aufgabe:

Ergänzen Sie die Klasse `Benutzer` um folgende Operatoren:

- Der Ausgabeoperator soll eine kommaseperierte Zeile (ohne die Nutzernummer) erzeugen.

[21] Wolf: Kap. 13

- Gleichheitsoperator: Zwei Instanzen sollen gleich sein, wenn ...
  - Falls beide Nutzernummern nicht null sind, werden die Nutzernummern verglichen.
  - Andernfalls sollen Name und Vorname verglichen werden.

Verwenden Sie die Operatoren.

- Implementieren Sie in der Nutzerverwaltung die Funktionen
  - `Benutzer* lesen(int nutzernummer)`, die die Nutzerdatei `nutzer.csv` Zeile für Zeile liest und die gelesenen Zeilen mitzählt. Wenn die Zeilennummer gleich der Nutzernummer ist, wird eine Nutzerinstanz am Freispeicher angelegt und an den Aufrufer zurückgegeben. Fall zu wenig Zeilen in der Datei sind oder die Datei gar nicht vorhanden ist, wird `nullptr` zurückgegeben.
  - `Benutzer* lesen(const Benutzer& nutzer)`, die in einer Schleife Nutzernummern von eins beginnend durchprobiert und über `lesen(int nutzernummer)` prüft, ob die gelesene Instanz gleich der übergebenen Instanz ist. Falls `lesen(int nutzernummer)` `nullptr` zurückgibt, bricht die Schleife ab und gibt gleichfalls `nullptr` zurück.
  - `anlegen` prüft über `lesen(const Benutzer& nutzer)`, ob der Benutzer schon im System ist. Falls ja, wird der entsprechende Fehlercode zurückgegeben, falls nein, wird die Datei `nutzer.csv` zum Anhängen geöffnet und über den *outstream operator* eine weitere Zeile in die Datei geschrieben.
- Testen Sie die Neuimplementierung.

5.3 Vererbung[22]

Motivation: Gemeinsamkeiten nur einmal implementieren: s. Abb. 3.

Definition der Mutterklasse: [`014/vererbung001/person.h`, W:347]

Ableitung von Kunde: [`014/vererbung001/kunde.h`, W:348]

In Kunde wird das "Mehr" des Kunden gegenüber der Person implementiert. Alle Eigenschaften von `Person` *erbt* `Kunde`.

Wie zu sehen ist, enthalten sowohl `Person` als auch `Kunde` eine Funktion mit der Signatur `void print() const`.
Dies nennt man *Überschreiben* von Methoden. Auf die überschriebene Methode der Mutterklasse kann direkt aus der Kindklasse zugegriffen werden:

```
void print() const {
   std::cout << "Kundennummer:" << get_kundennummer() << std::endl;
   Person::print();
   // statt:
   // std::cout << "Vorname     :  " << get_vorname() << std::endl;
```

[22] Wolf: Kap. 15
[23] Wolf: S. 346

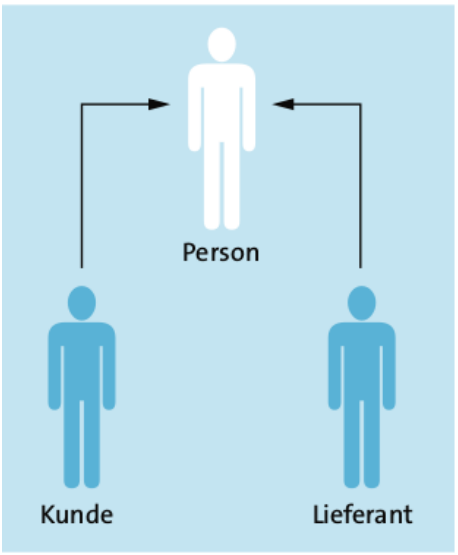


Abbildung 3: Vererbungsbeziehung[23]

```
    // std::cout << "Nachname    :  " << get_nachname() << std::endl;
}
```

Beim Instanzieren einer abgeleiteten Klasse werden sowohl der Konstruktor der abgeleiteten Klasse, als auch der Konstruktor der Basisklasse aufgerufen, hier per default der Standardkonstruktor. Durch eine explizite Angabe in der Initialisiererliste kann auch ein anderer Konstuktor der Basisklasse gewählt werden [Zeile 15].

Zugriffsrechte: Abbildung 4

Schlüsselwort	Eigene Klasse	Abgeleitete Klasse	Außerhalb
private	sichtbar	nicht sichtbar	nicht sichtbar
protected	sichtbar	sichtbar	nicht sichtbar
public	sichtbar	sichtbar	sichtbar

Abbildung 4: Sichtbarkeit bei der Vererbung[24]

Die abgeleitete Klasse ist immer zur Basisklasse typkompatibel (*is-a*), aber nicht umgekehrt: [`014/vererbung003/main.cpp`, W:359]

Aufgabe:

Ergänzen Sie die Bibliotheksanwendung um die Dateien `Medium.h/.cpp`:

- Leiten Sie von der Klasse `MediumBase` die Klassen `Buch` und `CD` ab.

[24] Wolf: S. 357

- Verteilen Sie die Attribute sinnvoll auf die Klassen.
- `Buch` und `CD` erhalten einen Konstruktor mit 4 Parameter.
- Ergänzen Sie jede Klasse um eine Methode `string format() const`, die den Datensatz als Zeichenkette in dem Format zurückgibt, wie er in der Mediendatei erwartet wird. Für die Umwandlung von `int` in `string` gibt es die `std::to_string`-Funktion.
- Für die Klasse `MediumBase` implementieren Sie eine Dummy-Ausgabe.
- Testen Sie die `format`-Funktion für alle drei Klassen im Hauptprogramm.

Polymorphismus

Zur Motivation dieses Kapitels implementieren Sie diese Aufgabe:

Aufgabe:

Ergänzen Sie die Bibliotheksanwendung:

- Ergänzen Sie die Medienverwalatung um eine Funktion

  ```
  int addMedium(const MediumBase& medium),
  ```

  die die neue `format()`-Methode zum Schreiben in die Datei verwendet. Auf die Duplikatsprüfung kann erst mal verzichtet werden.
- Verwenden Sie die Methode im Hauptprogramm, indem Sie einmal ein Buch, einmal eine CD übergeben.

Sie werden sehen, dass in der Datei nur immer der Dummy-Eintrag für ein *Medium* erscheint, obwohl Sie im Hauptptogramm konkrete Instanzen implementiert haben. An dieser Stelle wird nun ein Mechanismus benötigt, der zur Laufzeit die tatsächliche Klasse einer Instanz bestimmt und danach die passende Implentierung auswählt. Dieser Mechanismus nennt sich *Polymorphismus* (Vielgesichtigkeit).

Das polymorphe Überschreiben von Methoden erfolgt duch die *modifier* `virtual` in der Basisklasse und `override` in den abgeleiteten Klassen [`014/virtual02/main.cpp`, W:362].

Aufgabe:

Ergänzen Sie die Bibliotheksanwendung:

- Implementieren Sie die `format()`-Methode polymorph.
- Bringen Sie die neue `addMedium`-Methode zum Laufen.

Abstrakte Klassen und Methoden

Die Dummy-Implementierung in `MediumBase` macht nicht viel Sinn, da ja Instanzen von `MediumBase` auch nicht viel Sinn machen. `MediumBase` kapselt ja nur die Gemeinsamkeiten von `Buch` und `CD`. Die Gemeinsamkeit bezüglich der `format()`-Methode ist, dass sie in den abgeleiteten Klassen vorhanden ist. Die reine Existenz lässt sich mit einer Deklaration `virtual string format() = 0;` beschreiben. Es wird dem Compiler gesagt, dass die abgeleiteten Klassen diese Methode implementieren *müssen*. Als Konsequenz verhinder der Compiler, dass Instanten von `MediumBase` gebildet werden. Die `format()`-Methode in `MediumBase` nennt sich *abstrakte Methode*.

Eine Klasse, die mindestens eine abstrakte Methode hat, nennt sich *abstrakte Klasse*[25].

Aufgabe:

Ändern Sie `MediumBase` zu einer abstrakten Klasse ab.

Aufgabe zu `unique_ptr`:

- Ergänzen Sie die Definition der Klasse `MediumBase` um eine statische Methode (`static MediumBase* createMedium(signatur, autor, ...)`), die eine Instanz von `Buch/CD` im Freispeicher erzeugt. Innerhalb dieser Methode soll die Instanz in einem `unique_ptr` gehalten werden.
- Rufen Sie die Funktion im Hauptprogramm auf. Halten Sie dort die Instanz gleichfalls in einem `unique_ptr` und übergeben Sie die Instanz an `addMedium`.

6 Templates[26]

Templates bieten eine Form der generischen Implementierung. Man unterscheidet Funktionstemplates von Klassentemplates.

6.1 Funktionstemplates

Motivation: Identischer Code, der sich nur in einigen Typen unterscheidet
Beispiel: [`015/Templates001/main.cpp`, W:367].

Implementierung: [`015/Templates002/main.cpp`, W:368]

Umgang mit Ausnahmen, also Typen, die eine spezielle Implementierung erfordern:

```
std::string str1{"Hallo"}, str2{"Welt"};
std::cout << "Groesster Wert: "
  << bigNum( str1, str2 ) << std::endl;
```

[25] Wolf: S. 364
[26] Wolf: Kap. 15

Beispiel: [015/Templates004/main.cpp, W:371]

Templates mit mehreren variablen Typen: [015/Templates005/main.cpp, W:372f.]

Explizite Instanzierung eines Templates mit einem bestimmten Typ:
Beispiel: [015/Templates006/main.cpp, W:374f.]

Aufgabe:

In `util` ist die Funktion `toInt` definiert. Eine Umwandlung in `float`, `bool` oder `char` ließe sich in gleicher Weise implementieren.

- Schreiben Sie eine Templatefunktion `T toVal(const std::string& token)`, die die Umwandlung in analoger Weise vornimmt.
- Ersetzen Sie die Implementierungen in `toInt` und `toChar` durch die Verwendung von `toVal`. Beachten Sie, dass hier eine explizite Instanzierung nötig ist.

6.2 Klassentemplates

Definition: [015/SimpleCTemp/CTemp.h, W:376f.]

Instanzieren:

```
CTemp<double> dval;
CTemp<std::string> sval;
```

Für Templates wird erst dann Code gebildet und compiliert, wenn es für einen bestimmten Typ instanziert wird. Daher muss die Templatedefinition in einer Headerdatei stehen, die vor jeder Verwendung inkludiert werden muss. Während es bei normalen Funktionen oder Klassen vom Linker verboten ist, wenn diese mehrfach in Objektdateien vorkommen, verwirft er identische Instanzierungen.

6.3 Templates der Standardbibliothek

Container der Standardbibliothek[27] →`vector`, `map`, `array` (C++11)

Eine typische Verwendung von `std::map` finden Sie in Abb. 5.

Um über Container zu iterieren gibt es das Iteratorkonzept. Der Iterator ist ein spezieller Zeiger auf ein Element des Containers. Mit der Methode `begin()` erhält man den Iterator, der auf das erste Element des Containers zeigt, `end()` ist nach dem letzen Element positioniert. Der ++-Operator schiebt den Iterator auf das nächste Element. Eine Schleife über alle Elemente lässt sich daher so formulieren:

[27] http://www.cplusplus.com/reference/stl/

```
#include <map>
// definition
std::map<std::string,int> mymap;
// Element hinzuf"ugen / "uberschreiben
mymap["eins"] = 1;
// Element abfragen. Wenn nicht existent, wird es eingef"ugt.
int i = mymap["eins"];
// Element abfragen. Wenn nicht existent, wird eine Exception geworfen.
int j = mymap.at("drei");
// Abzahl der Elemente in der map
int a = mymap.size();
// Anzahl der Elemente mit einem bestimmten Schl"ussel (kann 0/1 sein)
int k = mymap.count("zwei");
```

Abbildung 5: Verwendung von `std::map`

```
std::map <std::string,std::string> phonebook;
// fill phonebook with data
std::map <std::string,std::string>::iterator mapIt;
for ( mapIt=phonebook.begin(); mapIt!=phonebook.end(); mapIt++)
        std::cout << mapIt->first << ": " << mapIt->second << std::endl;
```

Range-basierte For-Schleife und Typableitung

Aus vielen modernen Programmiersprachen bekannt (foreach)

- für C-Array
- für Container (u.a. `vector`, `map`, `initializer_list`)
- `for (typ var : container_var)`

Der Compiler weiß an vielen Stellen, was für ein Typ verwendet werden muss.

- Typisierung mit `auto`

  ```
  auto i = 5;

  for (auto var : container_var)
  ```

- Typisierung mit `decltype`

  ```
  int b;
  decltype(b) a;
  ```

`auto` steht immer für den Wert-Typ (keine Referenz). Soll eine Referenz verwendet werden, steht `auto&`.

Mit C++11 ist es damit sehr einfach geworden über diese Container zu iterieren.

```
for ( auto& mapIt: phonebook)
        std::cout << mapIt.first << ": " << mapIt.second << std::endl;
```

Aufgabe:

Ergänzen Sie die Bibliotheksanwendung um eine Ausleiheverwaltung. Die Ausleihen sollen (nicht persistent) in einer `std::map`[28]abgelegt werden.

- Legen Sie die Klasse `Ausleiheverwaltung` mit den internen Daten `std::map<std::string, Benutzer> ausleihen;` an. Der `string` soll die Signatur sein, der Benutzer ist derjenige, der das Medium gerade ausgeliehen hat.
- Schreiben Sie eine Methode `bool mediumIstAusgeliehen(const string& signatur)`, die prüft, ob der gegebene Schlüssel in der `map` vorhanden ist.
- Schreiben Sie eine Methode `void show()`, die über das Verzeichnis iteriert und alle ausgeliehenen Medien ausgibt. Beachten Sie, dass das Element einer `std::map` ein Schlüssel-Wert-Paar ist. Auf den Schlüssel wird mit `.first`, auf den Wert mit `.second` zugegriffen.
- Schreiben Sie eine Methode `int anlegen(string, const Benutzer& nutzer)`, die `mediumIstAusgeliehen` aufruft und die Ausleihe der Map hinzufügt.

  Anmerkung: `anlegen` haben wir jetzt zwei mal im System. Einmal als Funktion in der Nutzerveraltung, einmal als Methode der Ausleiheverwaltung. Zur besseren Unterscheidung kann für die Nutzerverwaltung ein Namensraum eingeführt werden.
- Ergänzen Sie das Hauptprogramm so, dass nun Ausleihen angelegt werden können.

Algorithmen der Standardbibliothek

Algorithmen der Standardbibliothek[29] → `for_each()`, `find()`, `find_if()`, `copy()`, `count()`, `count_if()`, `sort()`

Viele dieser Algorithmen benötigen eine Funktion als Parameter. Die kann eine „normale" Funktion, aber auch eine Lambdafunktion sein.

6.4 Lambda-Funktionen

- Funktionen ohne Namen (anonym)
- Lassen sich dort einsetzen, wo Funktionszeiger oder Funktoren (ausführbare Objekte, das sind Instanzen von Klassen, die den `()`-Operator implementieren) vorkommen.

Syntax:

```
[*1](*2){*3}
*1: Bindung an den lokalen Kontext
  []: Keine Bindung
  [=]: Alle Werte werden kopiert (Schnappschuss).
  [&]: Alle Werte werden referenziert.
```

[28]Nützliche Informationen zur Standardlibrary findet man unter http://cplusplus.com
[29]http://www.cplusplus.com/reference/algorithm/

```
*2: Laufzeitparameter
*3: Implementierung
```

Bindung an den Kontext

Sollen Variablen aus dem aktuellen Kontext in die Lambdafunktion hineingenommen werden, müssen diese in den eckigen Klammern angegeben werden. Mit einem vorangestellen `&` wird die Variable als Referenz übergeben, sonst wird sie kopiert. Sollen alle Variablen als Referenz gebunden werden, steht `&` allein in der eckigen Klammer, ein = allein kopiert alle Variablen.

Aufgabe:

Ergänzen Sie die Klasse `Ausleiheverwaltung` um Funktionen, die die Algorithmen der Standardbibliothek verweden.

- `for_each`: Eine Funktion `speichern()` soll die Elelemte der Medienmap in eine Datei schreiben. Öffnen Sie vorher die Datei `ausleihen.csv` zum Schreiben und definieren Sie eine Lambdafunktion, die als Parameter das Schlüssel-/Wertpaar nimmt und Signatur und Nutzernummer kommasepariert in die die Datei schreibt.
- Ergänzen Sie den Konstuktor, der bei einem Vorhandensein dieser Datei diese Zeile für Zeile ausliest, über `Nutzerverwaltung::lesen(int)` die zugehörige Nutzerinstanz holt und die Map aufbaut.
- `count_if`: Eine Funktion `countMunchen()`, die die Anzahl der entliehenen Medien an Münchner Benutzer ausgibt.
- `for_each, sort, copy`: `nutzerAusgeben()` soll mit `for_each` eine Liste (`vector`) derjenigen Benutzer erstellen, die Medien ausgeliehen haben. Diese Liste wird dann sortiert. Da für `Benutzer` kein Kleiner-Operator definiert ist, bekommt das `sort` eine Prädikatfunktion übergeben, die Name, Vorname, Ort zu einer langen Zeichenkette zusammenhängt und vergleicht. Die sortierte Liste wird dann über `copy` ausgegeben. Das Kopierziel ist ein output stream iterator[30]

7 Muster

Jedes Muster beschreibt ein in unserer Umwelt beständig **wiederkehrendes Problem** und erläutert **den Kern der Lösung** für dieses Problem, so daß Sie diese Lösung **beliebig oft anwenden können**, ohne sie jemals ein zweites Mal gleich auszuführen. [31]

Ein Software-Architektur-Muster beschreibt ein bestimmtes, in einem speziellen Entwurfskontext **häufig auftretendes Entwurfsproblem** und präsentiert ein erprobtes

[30] http://www.cplusplus.com/reference/iterator/ostream_iterator/algorithm/ (3.5.2019)
[31] Christopher Alexander et al., 1977

generisches **Schema zu seiner Lösung**. Dieses Lösungsschema spezifiziert die **beteiligten Komponenten**, ihre **jeweiligen Zuständigkeiten, ihre Beziehungen** untereinander und die **Art und Weise, wie sie kooperieren**.[32]

Jede Muster-Form enthält bestimmte Grundelemente:

- Problem
- Lösung
- Beispiel

Der Klassiker: „Gang of Four"[33]

Die Erwartungen:

- Hilfestellung zur Problemlösung
- Wiederverwendung der Problemlösungen
- Reduzierung der Abhängigkeiten
- Software wird flexibel, erweiterbar, wiederverwendbar
- Kommunikationsbasis sowohl in der Design-Phase als auch in der Realisierungs-Phase

Warnung: Software ist nicht dann gut, wenn sie möglichst viele Muster enthält. Muster sind kein Selbstzweck.

7.1 Fabrik

Problem

Die Erzeugung von Objekten hängt oftmals von verschiedenen Umstanden ab. So kann eine Klasse verschiedene Unterklassen haben. Und in jeder Klasse kann es verschiedene Konstruktoren geben.

Lösung

Eine Lösung des Problems kann durch die Implementierung eines „Objekterzeugers", einer Fabrik, sein. Je nach Komplexität des Problems kann diese Fabrik eine einfache statische Methode, ein Objekt oder eine Objekthierarchie sein.

Beispiel

Ein einfaches Beispiel für eine Fabrik haben wir im Abschnitt 5.3 mit der Methode `createMedium` bereits implementiert.

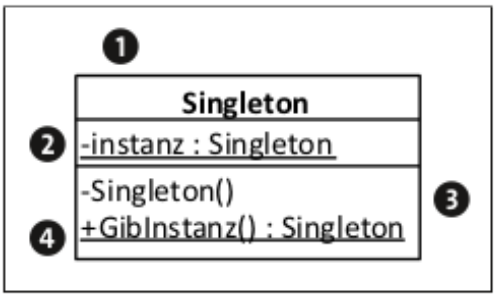
[32] Frank Buschmann et al., 1996
[33] Gamma et al. 1994

7.2 Singleton

Problem

Werden (z.B.) Systemresourcen über Klassen verwaltet, ist es nötig, den Zugriff dergestalt zu regulieren, dass an genau einer Stelle der Zustand der Resource bekannt ist. Während man in früheren Zeiten globale Systemvariablen dafür verwendet hat, bietet sich in einer objektorientierten Umgebung das *Singleton* dafür an.

Lösung



Nr.	Erläuterung
❶	Der Name der Klasse, hier exemplarisch »Singleton« benannt
❷	Der eigentliche Speicher für das Objekt ist ein Klassenattribut. Es ist privat, weil nur die *Singleton*-Klasse selbst es zuweisen darf.
❸	Der private Konstruktor verhindert, dass die *Singleton*-Klasse außerhalb der *Singleton*-Klasse selbst instanziiert werden kann.
❹	Clients greifen auf das *Singleton*-Objekt über diese öffentliche Klassenmethode zu.

Anmerkungen:

- Die Einmaligkeit bezieht sich auf den Speicherbereich. Laufen auf einem System mehrere Instanzen eines Programms, gibt es auch mehrere Singleton-Instanzen.
- Das Singtон ist nicht threadsicher. In einer *multi-threaded*-Umgebung müssen die Zugriffe auf die Instanz mit Semaphoren o.ä. abgesichert werden.

Beispiel

Hier sagen ein paar Zeilen Code mehr als viele Worte:

```
class Singleton {
  private:
    static Singleton* theInstance = nullptr;
    Singleton() {
      // construct an instance
    }
  public:
    Singleton* getInstance() {
      if (! Singleton::theInstance) Singleton::theInstance = new Singleton();
      return Singleton::theInstance;
    }
}
```

Anmerkungen zum Code:

- In modernem C++ werden die nackten Zeiger durch *smart pointer* ersetzt.
- Deklaration und Definition sind wie gewohnt auf die .cpp/.h-Datei zu verteilen.

Aufgabe:

Um in einem System den fachlichen vom technischen Kontext zu trennen, sollen die Schnittstellen der fachlichen Klassen keine technischen Aspekte enthalten. Um aber innerhalb der Implementierung auf technische Aspekte wie die Ausgabe von Systemmeldungen und Transaktion zugreifen zu können, können diese als Singleton-Instanzen bereit gestellt werden.

- Ergänzen Sie die Büchereianwendung um die Klasse `Systemmelder`, die als Singleton zu Implementieren ist. Als „Nutzschnittstelle" enthalte diese Klasse die Methode `void melde(string)`, die die übergebene Zeichenkette auf der Konsole ausgibt.
- Ergänzen Sie das Absetzen einer Meldung in der Funktion `addMedium`, falls es sich um ein Duplikat handelt.

7.3 Strategie[34]

Problem

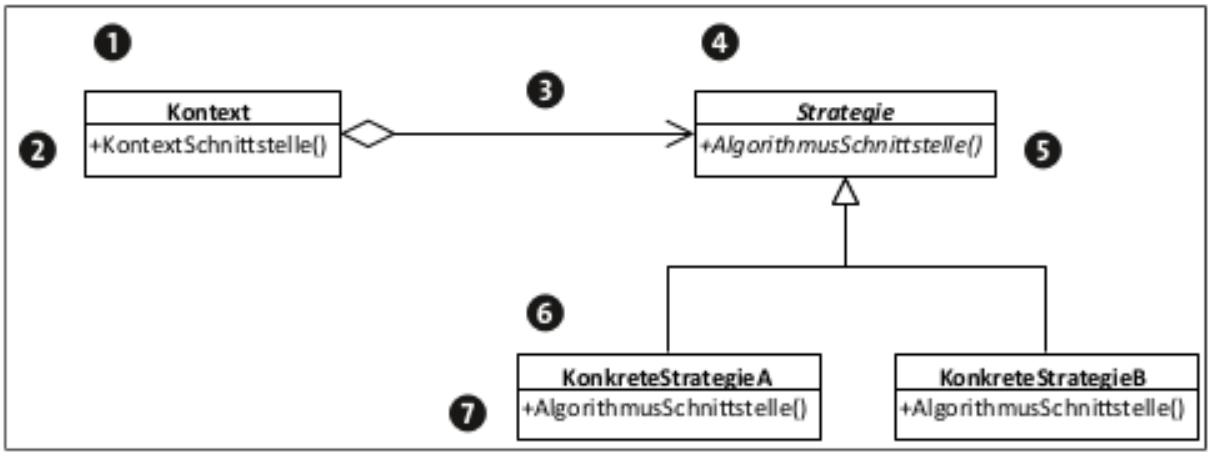
Manchmal führen viele Wege zum Ziel, und Wege in der Softwareentwicklung sind Algorithmen, also Verhalten. Welcher Algorithmus zum Ziel kommt, entscheidet vielleicht der Anwender, oder der Klient wählt den optimalen Algorithmus aufgrund der konkreten Aufgabenstellung aus.

Einzelne Algorithmen mögen ihre spezifischen Vor- und Nachteile haben. Vielleicht liefert ein Algorithmus besonders genaue Ergebnisse, benötigt aber viel Rechenzeit und Arbeitsspeicher, während ein anderer Algorithmus eine gute Näherung mit wenig Ressourceneinsatz errechnet. Oder Algorithmen unterscheiden sich aufgrund ihres Outputs, den sie mit demselben Input generieren.

[34] Geirhos 2015, S. 343ff.

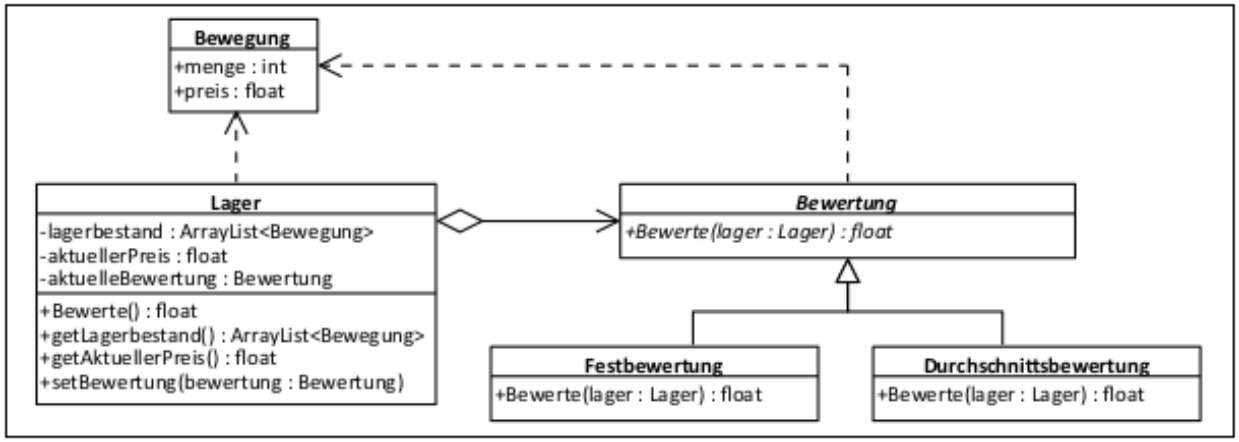
Wie auch immer ein Algorithmus aussieht und wer ihn auch immer auswählen mag: Eine Software muss natürlich alle infrage kommenden Algorithmen implementie- ren. Das Strategiemuster hilft dabei, eine solche Familie von alternativen, austauschbaren Algorithmen zu definieren und einer Anwendung zur Verfügung zu stellen. Wie so häufig ist auch hier das Ziel, dass die Festlegung auf einen bestimmten Algorithmus erst zur Laufzeit erfolgen soll.

Lösung



Nr.	Erläuterung
❶	Die Klasse *Kontext* ist auch hier wieder die Klasse, innerhalb derer eine Verarbeitung stattfindet. Sagen wir, diese Klasse wäre eine Klasse, die für ein Warenlager steht.
❷	Der Kontext definiert eine *Schnittstelle*, also bestimmte Methoden, bei deren Ausführung ein Algorithmus benötigt wird. Im Beispiel könnte das die Methode `Bewerte` sein, und der Algorithmus wäre dann ein entsprechender Bewertungsalgorithmus.
❸	Nun gibt es aber nicht nur einen Algorithmus, sondern derer mehrere. Im Beispiel hat dafür schon der Gesetzgeber gesorgt, und so gibt es für die Bewertung von Lagerbeständen die Festbewertung, die Durchschnittsbewertung (in mehreren Varianten) und die Verbrauchsfolgebewertung. Der Kontext implementiert alle diese Verfahren nicht selbst, sondern kennt wiederum Klassen, die diese Verfahren implementieren.
❹	Diese Klassen implementieren alle die gemeinsame Schnittstelle *Strategie*.
❺	Diese Klasse wiederum deklariert eine gemeinsame Schnittstelle, die *Algorithmusschnittstelle*.
❻	Die *konkreten Strategieklassen* gibt es nun für jeden zu implementierenden Algorithmus, im Beispiel also für alle drei Bewertungsalgorithmen.
❼	Die Implementierung des jeweiligen Algorithmus steckt in den konkreten Methoden.

Beispiel



Hinweis: Um eine technische Strategie zu realisieren, wir dieses Muster gern mit dem Singleton kombiniert. In diesem Fall ist dann der *Kontext* eine technische Klasse, die der fachlichen Seite über das Singleton zur Verfügung gestellt wird.

Aufgabe:

Ergänzen Sie die Büchereianwendung um die Möglichkeit, Meldungen wahlweise an der Konsole oder im Log auszugeben:

- Schreiben Sie eine abstrakte Klasse `IMelder` mit `melden` als akstrakte Methode.
- Leiten Sie davon die Klassen `Konsolenmelder` und `Logger` ab, die die Meldung auf der Konsole ausgeben, resp. in eine Datei schreiben.
- Das Öffnen und Schließen der Datei erfolgt dadurch, dass der `ofstream` als Member von `Logger` definiert wird.
- Die Klasse `Systemmelder` bekommt nun einen `unique_ptr<IMelder>` als Member, dazu die Methode `void setMelder(unique_ptr<IMelder>)`.
- Die Methode `melden` von `Systemmelder` prüft, ob der Melder gesetzt wurde. Falls ja wird `melden` aufgerufen.
- Im Hauptptogramm wird schon mal ein `Systemmelder` instanziert und der gewünschte Melder gesetzt.
- Testen Sie den Code für den `Konsolenmelder` und den `Logger`.

8 Quellen

Alexander, Christoph et al.	A Pattern Language, Oxford Press 1977
Buschmann, Frank et al.	Pattern-Oriented Software Architecture. A System of Patterns, John Wiley 1996
Gamma, E., R. Helm, R. Johnson, J. Vlissides	Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley 1994
Geirhos, Matthias	Entwurfsmuster. Das umfassende Handbuch, Rheinwerk 2015
Grimm, Rainer	C++11 für Programmierer, 2014
Wolf, Jürgen	Grundkurs C++, 3. Auflage