NewElements

IT-Schulungen.com

C++ - Seminar

Grundkurs

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

Revision 292

*michael@wagnertech.de

Inhaltsverzeichnis

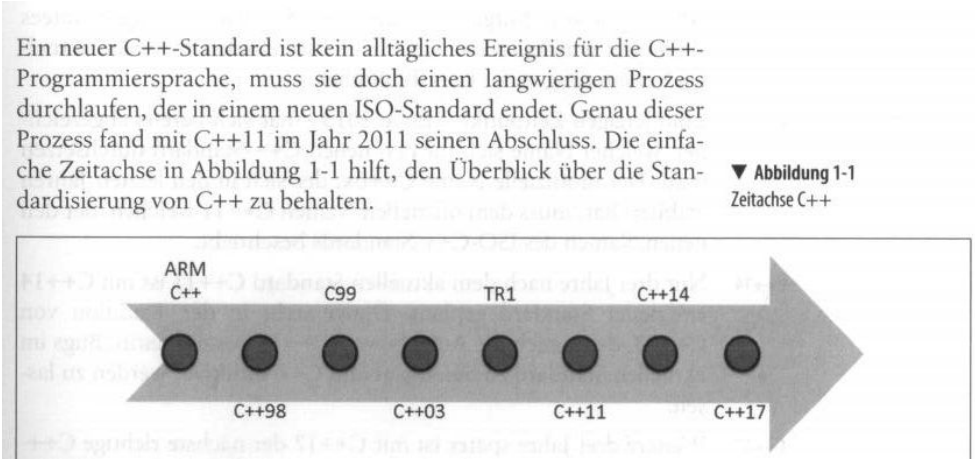

Abbildung 1: C++-Geschichte[2]

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 objektorietieret 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

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 Spachelemente 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
  - `std::cin`: Standard-Eingabestream

[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++

- 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 Basistypen[5]

C++ kennt nur wenige Basistypen. Die usprüngliche Idee war, alles „höherwertige" dem Anwender als Klasse zu überlassen. Dies hat aber in den 90er Jahren zu einem Wildwuchs an Klassenbibliotheken geführt, so dass im Jahr 1998 die Standardlibrary festgelegt wurde. Seit dieser Zeit gibt es beispielsweise eine Standardklasse für Zeichenketten: `std::string`.

Basistypen sind:

- Ganzzahltypen
- Varianten: `signed` und `unsigned`
- Fließkommazahlen
- `bool`

[5] Wolf: Kap. 3

[Variables and types[6]→Fundamental data types, W:48,50,59,87]

Zur Erzeugung von Konstanten kennt C++ drei Möglichkeiten:

- `#define`: Diese bereits aus C stammende Precompiler-Direktive führt zu einem direkten Suchen/Ersetzen zur Compilezeit. Nachteil: Nicht typicher
  `#define PI 3.1415`
- Konstante Typen (`const`): Laufzeitvariable, für die der Compiler zusichert, dass die nicht geändert wird. Nachteil: Über Zeigermanpulation kann jede Speicheradresse des Programms geändert werden.
  `const double pi=3.1415;`
- `constexpr`: Neu mit C++11. Variable wird typsicher bereits zur Compilezeit ersetzt.
  `constexpr double pi=3.1415;`

Die von einem Variablentyp benötige Speichergröße kann mit dem `sizeof`-Parameter abgefragt werden [`003/listing006.cpp`, W:63].

Bei Operationen und Zuweisungen unternimmt C++ implizite Typumwandlungen. Diese können zu Problemen führen, da Werte unter Umständen verfälscht werden. Normalerweise sind die Compiler so eingestellt, dass sie in diesem Fall Warnungen ausgeben.

Variablen können in C++ an verschiedenen Speicherorten angelegt werden:

- Im Lademodul: Variablen die außerhalb eines jeden Programmblocks (`{ ... }`) definiert werden, liegen im Lademodul. Sie bestehen während des gesamten Laufes des Programms.
- Auf dem Stapel (*stack*): Variablen, die innerhalb eines Programmblocks definiert werden, sind während dieser Programmblocks duchlaufen wird, auf dem Programmstapel vorhanden.
- Variablen können auch im Freispeicher (*heap*) liegen. Dazu ist das Schlüsselwort `new` notwendig. Für die Freigabe von Daten im Freispeicher ist der Entwickler selbst verantwortlich. Die Freigabe erfolgt mit `delete`. Da meist Instanzen von Klassen im Freispeicher angelegt werden, erfolgt die nähere Beschreibung hierfür in Kap. 6.

2.2 Arithmetische Operatoren[7]

Folgende mathematische Grundoperationen stehen in C++ zur Verfügung: `+ - * / %`

Kurzschreibweise für Operationen auf die Variable selbst: `+= -= *= /= %=`

Inkrement und Dekrement: `++ --`

[Operators[8], W:68f,72,73]

[6] https://www.cplusplus.com/doc/tutorial/variables/
[7] Wolf: Kap. 4
[8] https://www.cplusplus.com/doc/tutorial/operators/

2.3 Arrays und Zeichenketten[9]

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)

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[10]

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.

  Anmerkung: In C++ werden Variablennamen von einfachen Variablen üblicherweise in kleinen Buchstaben geschrieben. Die Namen können einen Unterstrich enthalten.
- Eine Prüfung der Eingabe erfolgt nicht, wir gehen von einem gutmütigen Benutzer aus.
- Geben Sie die Variablen schön formatiert wieder aus.

[9] Wolf: Kap. 6

[10] http://www.cplusplus.com/reference/string/string/

2.4 Referenzen und Zeiger[11]

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]

3 C++ Kontrollstrukturen

3.1 Verzweigungen[12]

Verzweigungen erfolgen anhand logischer Ausdrücke. Diese werden mit Vergleichsoperatoren erzeugt und mit logischen Operatoren verknüpft:

Vergleichsoperatoren: `< <= > >= == !=`

Logische Operatoren: `&& || !`

[Operators→Relational and comparison operators, Logical operators[13], W:92,100]

Anstatt eines Vergleichs kann ein logischer Ausdruck auch die Abfrage eines Zustands sein. So bedeutet beispielsweise der Ausdruck `cin.fail()`: „Ist die letzte Abfrage von der Tastatur schief gegangen?"

Für Verzweigungen stehen drei Sprachelemente zur Verfügung:

- `if - else if - else` [`005/listing003.cpp`, W:95]
- Bedingungsoperator `? :` [`005/listing004.cpp`, W:98]
- `switch` [`005/listing007.cpp`, W:103]

> Aufgabe:
>
> Ergänzen Sie die Eingaben von letzter Aufgabe mit Prüfungen. Im Fehlerfall machen Sie eine Ausgabe und beenden das Programm. Unterscheiden Sie dabei folgende Fälle:
>
> - Bei der Eingabe einer Zeichenkette ist ein möglicher Fehler, dass diese leer ist. Prüfen Sie dies mit der Funktion `.empty()` ab.
> - Bei der Zeichen- oder Integereingabe können Sie am Eingabestrom erkennen, ob die Eingabe erfolgreich war (`cin.fail()`).

[11] Wolf: Kap. 7
[12] Wolf: Kap. 5
[13] https://www.cplusplus.com/doc/tutorial/operators/

Anmerkung: Eine wirklich fehlertolerante Eingabe kann so nicht erreicht werden. Ein Problem stellt die teilweise richtige Eingabe, also ein "56rt", wo eine Ganzzahl erwartet wird. In diesem Fall würde 56 in die Ganzzahl übernommen, "rt" bliebe im Eingabestrom stehen und würde vermutlich bei der nächsten Eingabe zu einem Fehler führen. Im Prinzip ist hier ein zeilenweises Denken von nöten. Dies kann mit der Funktion `readline` erreicht werden, die bei der Dateibehandlung vorgestellt wird.

3.2 Schleifen[14]

In C++ gibt es folgende Schleifen:

- Zählschleife: `for` [005/listing010.cpp, W:112f]
- Schleife über den Inhalt eines Containers (*range for*): [006/listing002.cpp, W:125]
- Kopfgesteuerte Schleife: `while` [005/listing008.cpp, W:106f]
- Fußgesteuerte Schleife: `do - while` [005/listing009.cpp, W:109]

Aufgabe:

Da es nicht sehr benutzerfreundlich ist, wegen einer fehlerhaften Eingabe das ganze Programm abzubrechen, packen Sie jede Eingabe in eine Schleife nach folgendem Muster:

```
std::string signatur;
do {
  std::cout << "Bitte Signatur eingeben: ";
  std::cin >> signatur;
  if (signatur.empty()) {
    std::cout << "Die Signatur darf nicht leer sein!" << std::endl;
  }
} while (signatur.empty());
```

3.3 Funktionen[15]

Compiler und Linker

C++ ist eine compilierende Sprache. Der Compiler benötigt den Quellcode, der in ASCII-Dateien vorliegt. Aus den Quelldateien werden Objektdateien erzeugt, die mit einem Linker zu einem ausführbaren Programm verbunden werden (Abb. 3).

Neben ausführbaren Programmen können Objektdateien auch zu statischen oder dynamischen Bibliotheken gebunden werden. Eine statische Bibliothek muss zu jedem Programm, das sie nutzt,

[14] Wolf: Kap. 5
[15] Wolf: Kap. 8
[16] Wolf: S. 20.

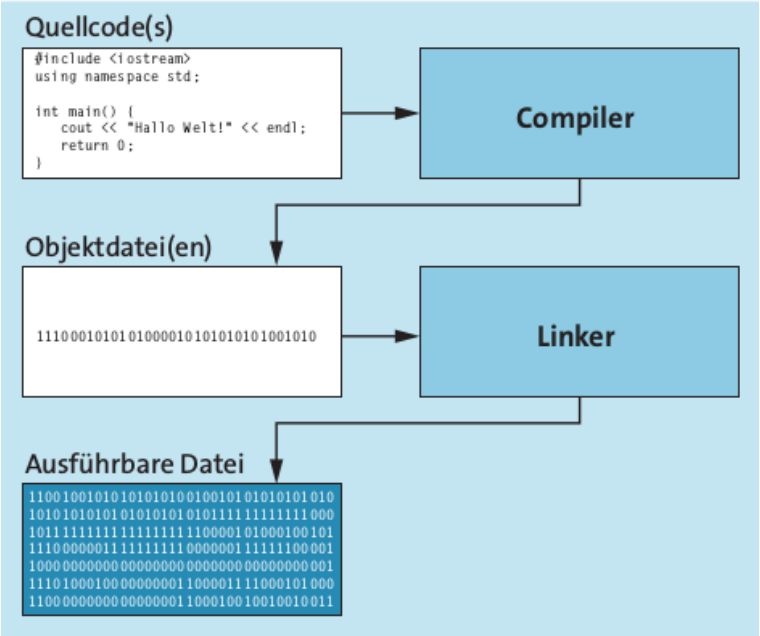

Abbildung 3: Vom Quellcode zur ausführbaren Datei[16]

gebunden werden. Eine dynamische Bibliothek wird zur Laufzeit zum Programm gebunden. Diese muss daher zur Laufzeit als Datei im System vorliegen.

Deklaration und Definition

Der Einstieg für ein ausführbares Programm ist die `main()`-Funktion. Die darf daher nur einmal vorkommen, in Bibliotheken gar nicht.

Mit selbstdefinierten Funktionen lässt sich ein Programm strukturieren. Damit eine Funktion verwendet werden kann, muss sie bekannt sein. Im einfachsten Fall wird sie in der selben Datei zuvor definiert: [`008/listing001.cpp`, W:151f]

Anmerkung: Das `void` ist hier der „Typ" des nicht vorhandenen Rückgebewerts. Wird tatsächlich ein Wert an das rufende Programm zurückgegeben, so steht statt `void` der Typ des Rückgabewerts.

Um größere Programme zu strukturieren, wird der Programmcode auf mehrere Dateien aufgeteilt. Findet sich die Implementierung der gerufenen Funktion in einer anderen Datei (oder schicht weiter unten in derselben Datei), muss die Funktion zuvor *deklariert* werden: [`008/listing002.cpp`, W:151]

Ändert sich aber die Signatur (z.B. Anzahl der Parameter) einer Funktion, müsste die Implementierung und mit ihr jede Deklaration konsistent geändert werden. Da dies sehr fehleranfällig ist, werden die Deklarationen (später auch Klassendeklarationien) üblicherweise in eine *Headerdatei* geschrieben, die überall dort inkludiert wird (Präprozessor-Direktive `#include`), wo die Funktion (später Klasse) verwendet wird. Um mehrfaches Inkludieren von Headerdateien zu vermeiden, enthalten Headerdateien eine `#ifndef - #define`-Sequenz:

```
// myheader.h
#ifndef MYHEADER_H_
#define MYHEADER_H_

// Der eigentliche Quellcode f"ur die Headerdatei myheader.h

#endif
```

> Aufgabe:
>
> Verteilen Sie das Beispiel `008/listing001.cpp` auf zwei cpp-Dateien (und eine Headerdatei).

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 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[17]:

```
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[18]:

```
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';
}
```

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

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

[17] s. auch Wolf: Kap. 8.2.1
[18] s. auch Wolf: Kap. 8.4

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[19]:

- Schreiben Sie ein Programm mit zwei Funktionen, mit denen Sie die Fläche und den Umfang eines Rechtecks berechnen. Die Fläche ermitteln Sie mit Länge x Breite und den Umfang mit 2 x Länge + 2 x Breite.

Ergänzung:

- 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.

[19] Wolf: S. 178ff.

3.4 Namensräume[20]

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 6) 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:

- 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.

[20] Wolf: Kap. 9.2

4 Ein- und Ausgabe von Dateien[21]

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;
}
```

- Für das blockweise Lesen und Schreiben stehen die Streammethoden `read` und `write` zur Verfügung.[22]

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)
```

[21] Wolf: Kap. 17
[22] Wolf: Kap. 17.3.6

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.

5 Fehlerbehandlung[23]

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.

[23] Wolf: Kap. 16

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

- 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.

6 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 3.4. 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".

6.1 Klassen[24]

Klassendefinition:[25]

```
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`[26]

Die Definition der Klassenmethoden erfolgt in der zugehörigen `cpp`-Datei:
`listings/011/automat1/automat.cpp`[27]

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
```

[24] Wolf: Kap. 11
[25] Wolf: S. 238.
[26] Wolf: S. 239.
[27] Wolf: S. 240.

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*[28]):

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

Ergänzen Sie die Bibliotheksanwendung

- Erstellen Sie eine Klasse `Medium` mit den entsprechenden Bestandteilen und Defaultwerten. In den objektorientierten Sprachen werden Klassen üblicherweise in gleichnamige Dateien (`Medium.cpp/.h`) gelegt.
- Ergänzen Sie die Klasse `Medium` um einen Konstruktor, der die einzelnen Bestandteile als Parameter übernimmt.
- Schreiben Sie zu jedem Datenelement einen *getter* (z.B. `string getSignatur() const;`, hier kann die Entwicklungsumgebung helfen!)
- Ergänzen Sie das Modul *Medienverwaltung* um eine Funktion `int addMedium(const Medium& medium)`, die die bereits bestehende Methode verwendet.
- Ergänzen Sie das Hauptprogramm um die Instanzierung eines Mediums auf dem Stapel und einen Aufruf dieser Methode.

Move vs. Copy

Problem: C++ kopiert viel zu viel:

- Objekte werden in die Container kopiert

  Hat das Objekt nach dem Kopiervorgang noch eine Verwendung?

[28] Wolf: S. 291f.

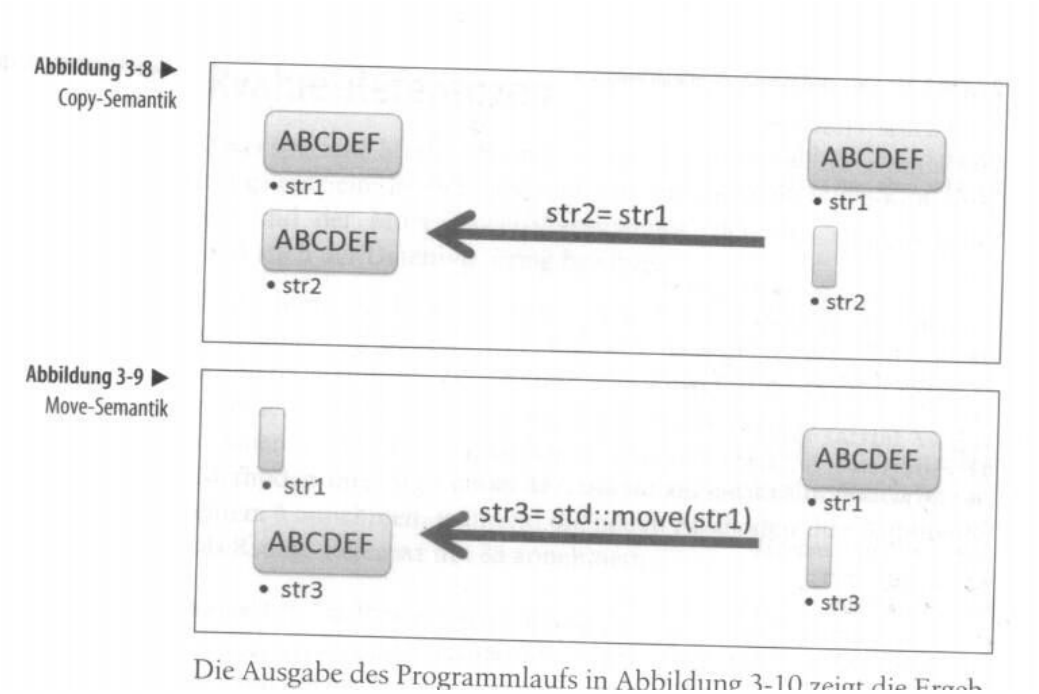

Abbildung 4: Move vs. Copy[29]

- Anders in Java, C#, Visual C++:

  Referenzen werden kopiert + Garbage-Collector

Move ermöglicht performanteres Übertragen von Objekten: Abb. 4.

- Wann wird move verwendet?

  explizit: `a = std::move(b);`

  implizit: Rechts steht ein Rvalue

- Was ist ein Rvalue?
  - Ein Ausdruck, der nur rechts vom Gleichheitszeichen stehen darf
  - Ausdruck ohne Name: `a = MyObject();`

Die Syntax für Move-Konstruktor und Move-Zuweisung:

```
MyClass(MyClass&&) { ... }
MyClass& operator=(MyClass&&) { ... }
```

[29] Grimm: S. 32

6.2 Operatoren[30]

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 `Medium` um einen Ausgabe-, einen Eingabeoperator sowie den Gleichheitsoperator.

- Der Ausgabeoperator soll genau so eine Zeile erzeugen, wie sie in die Datei `medien.csv` geschrieben wird.
- Der Eingabeoperator füllt die privaten Daten einer Instanz. Dazu wird die Zeichenkette in einen `stringstream` umgewandelt, danach Element für Element mit `getline(STREAM, STRING, SEP);` vom Stream geholt.
- Zwei Instanzen sollen gleich sein, wenn ihre Signaturen gleich sind.

Verwenden Sie die Operatoren.

- Ergänzen Sie die Klasse `Medium` um einen Konstruktor, der eine csv-Zeile als Parameter übernimmt und den Eingabeoperator verwendet.
- Ergänzen Sie die Medienverwaltung um eine Funktion `bool isSignatureInFile(const Medium& medium)`, die für die Duplikatsprüfung Zeile für Zeile liest, ein Medium erzeugt und die Signatur durch Instanzvergleich (==) geprüft.
- Ergänzen Sie die Medienverwaltung um eine Funktion `int addMedium(const Medium& medium)`, die
  - die Funktion `isSignatureInFile` nutzt,

[30] Wolf: Kap. 13

– die übergebene Medium-Instanz unter Verwendung des Ausgabeoperators an die Datei anhängt.

- Testen Sie die Neuimplementierung.

6.3 Vererbung[31]

Motivation: Gemeinsamkeiten nur einmal implementieren: s. Abb. 5.

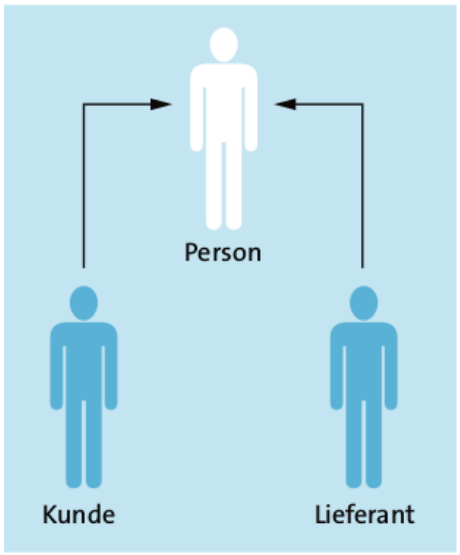

Abbildung 5: Vererbungsbeziehung[32]

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

[31] Wolf: Kap. 15
[32] Wolf: S. 346

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].

Ein reines „durchschleifen" von Konstruktoren der Basisklasse, kann seit C++11 auch abgekürzt werden:

```
class Basis {
public:
   Basis(int ival) { cout << ival << endl; }
   Basis(string s) { cout << s << endl; }
   Basis() { cout << "STD" << endl; }
};

class Erbe : public Basis {
public:
   // Vererbung der Konstruktoren
   using Basis::Basis;
   Erbe(float fval) { cout << fval << endl; }
};

// Verwendungsbeispiele
Erbe s1(222.222f);  // Erbe::Erbe(float)  + Basis::Basis()
Erbe s2(456);       // Basis::Basis(int)
Erbe s3("Hi Dad");  // Basis::Basis(string)
```

Zugriffsrechte: Abbildung 6

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

Abbildung 6: Sichtbarkeit bei der Vererbung[33]

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 `MediumHira.h/.cpp`:

- Leiten Sie von der Klasse `MediumBase` die Klassen `Buch` und `CD` ab.
- Verteilen Sie die Attribute sinnvoll auf die Klassen.
- `Buch` und `CD` erhalten einen Konstruktor mit 4 Parameter.

[33] Wolf: S. 357

- Ergänzen Sie jede Klasse um eine Methode `string format() const`, die den Datensatz als Zeichenkette in dem Format zurückgibt, wie er in der Eingabedatei 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*[34].

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`.

7 Templates[35]

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

7.1 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;
```

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.]

[34] Wolf: S. 364
[35] Wolf: Kap. 15

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.

7.2 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.

7.3 Templates der Standardbibliothek

Container der Standardbibliothek[36] → `vector`, `map`, `array` (C++11)

Eine typische Verwendung von `std::map` finden Sie in Abb. 7.

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 letzten Element positioniert. Der `++`-Operator schiebt den Iterator auf das nächste Element. Eine Schleife über alle Elemente lässt sich daher so formulieren:

```
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;
```

[36] 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 7: Verwendung von `std::map`

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 die Datei nicht jedes mal auf's Neue lesen zu müssen, sollen die Daten in einer `std::map`[37]abgelegt werden. Dazu bekommt die Medienverwaltung jetzt interne Daten, wird vom Funktionsmodul zur Klasse.

- Legen Sie die Klasse `MedienverwaltungClass` mit den internen Daten

  `std::map<std::string, Medium> medium_map;` an.

- Schreiben Sie eine Methode `load()`, die die Datei einliest und das Verzeichnis füllt. Als Schlüssel soll dabei die Signatur dienen, als Wert eine Instanz von `Medium`. Diese Funktion soll im Konstruktor aufgerufen werden.

  Nehmen Sie die Implementierung von `bool isSignatureInFile(const Medium& m)` als Vorlage.

- Schreiben Sie eine Methode `bool checkDuplicate(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 gespeicherten 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 addMedium(const Medium& medium)`, die `checkDuplicate` aufruft und das Medium an die Datei anhängt, danach das Verzeichnis mit `load` aktualisiert.

- Verwenden Sie den neuen Code im Hauptprogramm.

Algorithmen der Standardbibliothek

Algorithmen der Standardbibliothek[38] → `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 ein Funktionsobjekt sein. Ein Funktionsobjekt (Funktor) ist eine Klasse, die den `()`-Operator implementiert:

```
class MyFunctor {
private:
        int internal;
public:
        MyFunctor(int i) :internal(i) {}
        bool operator()(int i) { return i<internal; }
}
```

Mit C++11 kommt eine neue Möglichkeit, Funktionen an die Algorithmen zu übergeben: Die Lambdafunktionen.

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

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

Syntax:

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

Aufgabe:

- Compilieren und analysieren Sie das Beispiel `anhang_a/lambda.cpp`
- Ergänzen Sie das Beispiel um einen Anufruf mit einem Funktionszeiger und einem Funktor.

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 `MedienverwaltungClass` um Funktionen, die die Algorithmen der Standardbibliothek verweden. Zur Vorbereitung soll die Funktion `load()` nicht nur die Medienmap, sondern auch einen `vector` füllen, dessen Elemente die aus der Datei gelesenen Medien sind.

- `count_if`: Eine Funktion `countBooks()`, die die Anzahl der enthaltenen Bücher ausgibt.
- `for_each`: Eine Funktion `show1()` soll die Elelemte der Medienmap ausgeben.
- `copy`: Eine Funktion `show2()`, die die Elelemte des Medienvektors ausgibt. Das Kopierziel ist ein output stream iterator[39]
- `sort`: Eine Funktion `show_sort_author()`, die die Medien nach `autor` sortiert ausgibt:
  - Schreiben Sie eine Funktion
    `bool comp_media(const Medium& m1, const Medium& m2)`
    die `true` zurück gibt, falls `autor` von `m2` größer ist.
  - Sortieren Sie den Medienvektor mit der `sort`-Funktion.
  - Die Ausgabe erfolgt mit der Funktion `show2()`.
- `count`: Eine Funktion `checkDuplicate(const Medium&)`, die den Medienvektor durchsucht.
- `find_if`: Eine Funktion `checkDuplicate1(const string&)`, die die Medienmap durchsucht. Die Prädikatfunktion bekommt beim Aufruf ein `pair<string,Medium>` als Parameter. Der Schlüssel des Parameters (`.first`) muss dann mit der Signatur verglichen werden. Für die Implementierung gibt es zwei Möglichkeiten:

- Ein Funktor, der die Signatur im Konstruktor mitgegeben bekommt
- Eine Lambdafunktion, die die Signatur aus dem Kontext übernimmt

8 Quellen

Grimm, Rainer	C++11 für Programmierer, 2014
Wolf, Jürgen	Grundkurs C++, 3. Auflage

[39] http://www.cplusplus.com/reference/iterator/ostream_iterator/algorithm/ (3.5.2019)