NewElements

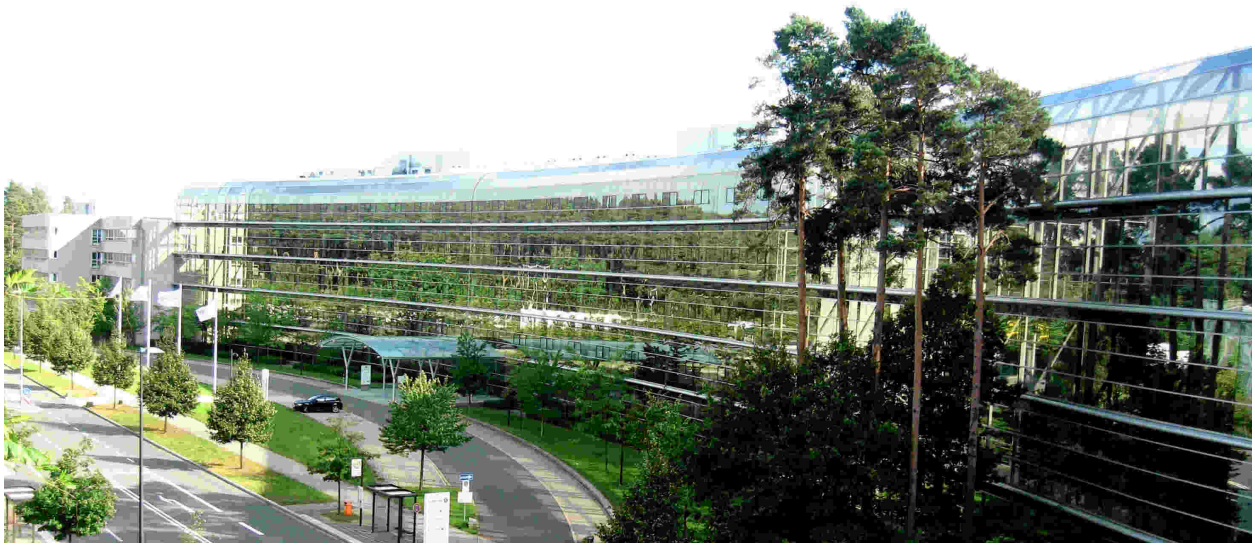
IT-Schulungen.com

C++11 - Seminar

Update Modernes C++

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

Revision 306



*michael@wagnertech.de

Inhaltsverzeichnis

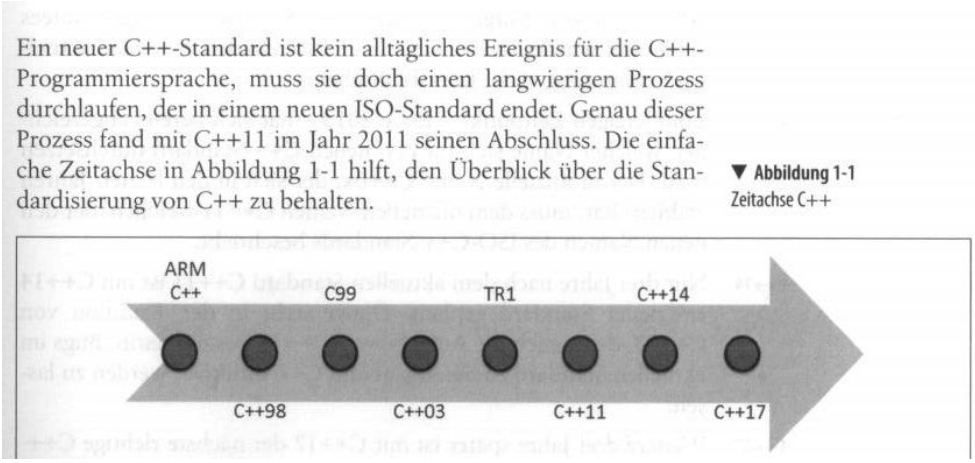


Abbildung 1: C++-Geschichte[2]

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

- 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

2 Beispielprogramm Bücherei

Als Beispielprogramm für diesen Kurs wird eine „Büchereiverwaltung" implementiert. Um nicht bei Null beginnen zu müssen, steht bereits eine einfache Implementierung des Geschäftsvorfall „Neues Medium anlegen" zur Verfügung. Diese Implementierung enthält folgende Dateien:

- `Bucherei.cpp`: Das Hauptprogramm
- `MedienUI.h/.cpp`: Funktionen zur Ein-/Ausgabe
- `MedienverwaltungClass.h/.cpp`: Bearbeitung des Geschäftsvorfalls
- `MediumHira.h/.cpp`: Medium-Klasse (mit Vererbung)
  `MediumBase`←`Buch, CD`
- `util.h/.cpp`: Hilfsfunktionen

[3] Grimm: S. 7

Aufgabe:

- Legen Sie in VisualStudio ein Projekt *Bucherei* an.
- Kopieren Sie die Dateien in das Projekt (`http://wagnertech.de/public/cpp/Bucherei.tgz`).
- Testen Sie den Code.

3 Neuerungen in der Kernsprache

Anmerkung: Wenn ein Codebeispiel mit Ihrem Compiler nicht funktioniert, können Sie den Wandbox[4]-Compiler verwenden.

3.1 Usability

Konstanten

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;`

Mit C++17 kommt `constexpr-if` hinzu:

```
if constexpr(any_compile_time_boolean_expression)
{
  // this is only compiled, if any_compile_time_boolean_expression is true
  int i = 5;
}
else
{
  double i = 5.;
}
```

Dies findet Anwendung mit den neuen Möglichkeiten Typinformationen zur Compilezeit abzufragen.

[4] https://wandbox.org/

Aufgabe:

Ersetzen Sie die `const`-Ausdrücke in `util.h` durch `constexpr`.

Anmerkung: Bei der Implementierung mit `const` werden Nur-Lese-Variable in jede Objektdatei hineincompiliert. Bei der Verwendung von `constexpr` entfällt dies, da die Ersetzung zur Compilezeit geschieht.

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&`.

`decltype` hingegen liefert auch Referenztypen. Mit C++14 steht das Konstrukt `decltype(auto)` zur Verfügung, das wie `auto` verwendet wird und auch Referenztypen liefert:[5]

```
auto i            = 5;     // int
int& iref         = i;     // int&
auto c            = iref; // int
auto& d           = i;     // int&
decltype(iref) e = i;     // int&
decltype(auto) f = iref; // int&
```

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;
```

In Iteratorschreibweise ohne `auto` lautete derselbe Code:

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

[5] https://en.wikipedia.org/wiki/C%2B%2B14 (27.8.2018)

Aufgabe:

Ersetzen Sie in `MedienverwaltungClass::show()` die entsprechende Schleife durch ein *range for*.

nullptr

Bisher wurde `NULL` für einen Zeiger benutzt, der auf keinen gültigen Speicher zeigt. `NULL` ist aber als Alias für die Zahl `0` definiert. Daher ist `NULL` vom Typ `int`. Mit C++11 war es das Ziel zwischen Zeigern und Zahlen unterscheiden zu können. `nullptr` wurde daher im Sprachkern definiert und ist vom Typ `void*`.

Aufgabe:

Ersetzen Sie die `NULL`-Ausdrücke im Projekt durch `nullptr`.

Vereinheitlichte Initialisierung

C++ 11 hat eine einheitliche Initialisierungs-Syntax für

- Strukturen
  Klassen, die ausschließlich `public` *members* haben und keine Konstruktoren aufweisen.
- C-Arrays
- Container
- Im Konstruktor
- Arrays auf dem Heap

[`Kernsprache/uniformInitialisation.cpp`]

3.2 Diverse Neuerungen in der Klassendefinition

- Die Initialisierer-Liste: ein neuer Container.
  - Verwendung in Konstruktoren
  - Deklaration: `MyClass(std::initializer_list<typ> data);`
  - Verwendung: `MyClass myClass{ele1, ele2, ele3};`
  - Beispiel: [`TourDeC++11/initializerListConstructor.cpp`][6]

[6] Grimm: S. 20.

- Delegation von Konstruktoren:

  Mit der „:"-Syntax kann ein anderer Konstruktor aufgerufen werden:
  .[TourDeC++11/delegatingConstructor.cpp][7]

- Vererbung von Konstruktoren: 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)
```

- Direktes Initialisieren von Klassenelementen: `int x = 5;` (wie bei Java)
  .[TourDeC++11/classMemeberInitializer.cpp][8]
- Steuerung der compilergenerierten Konstruktoren+Operatoren: `default, delete`
  .[TourDeC++11/defaultedDeletedMethods.cpp][9]
- Steuerung der Vererbung von Methoden: `override, final` (wie bei Visual-C++/C#)

Aufgabe:

Ergänzen Sie das Beispiel `initializerListConstructor.cpp` in der Weise, dass Sie folgende Konstruktoren zur Verfügung haben:

- Standardkonstruktor
- Initialisiererliste für Integer
- Konstruktor mit zwei Integer

Prüfen Sie nun, welche Instanzierung welchen Konstruktor aufruft:

- `Klasse k1;`
- `Klasse k2{};`
- `Klasse k3(3,4);`

[7] Grimm: S. 22.
[8] Grimm: S. 24f.
[9] Grimm: S. 27f.

- `Klasse k4{3,4};`

Entfernen Sie nun nach und nach die Konstruktoren, damit Sie erkennen, welche dieser Instanzierungen auch auf andere Implementierungen ausweichen können.

3.3 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.
*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.

Ab C++14 können auch Variablen explizit aus dem Kontext vorbelegt werden, dabei kann auch die `move`-Funktion verwendet werden:

```
auto lambda = [value = std::move(ptr)] {return *value;};
```

Aufgabe:

Ändern Sie `lambda.cpp` so ab, dass der Divisor variabel ist. Der Divisor soll nun

- durch eine Variable aus dem Kontext bestimmt werden,
- durch Zuweisung einer Zahl in der Kontextdefinition belegt werden.

Generische Lambdas (C++14)[10]

In C++11 mussten die Typen der Parameter deklariert werden. Mit C++14 können auch die Parameter mit `auto` allgemein gehalten werden.

```
auto lambda = [](auto x, auto y) {return x + y;};
```

[10] https://en.wikipedia.org/wiki/C%2B%2B14 (27.8.2018)

Dieser Code ist äquivalent zu einem ausführbaren Klassentemplate (Funktor):

```
struct unnamed_lambda
{
  template<typename T, typename U>
    auto operator()(T x, U y) const {return x + y;}
};
auto lambda = unnamed_lambda{};
```

Generische Lambdas sind innerhalb generischem Code, also bei der Implementierung von Templates nützlich. Auch der Umgang mit `variant` wird dadurch einfacher (s. Kap. 4.3).

> Aufgabe:
>
> Verwenden Sie in `MedienverwaltungClass.cpp` in den Funktionen `countBooks`, `showBooks` und `getMedium` Lambdafunktionen.

bind und function

`bind` macht aus bestehenden Funktionsobjekten neue, indem es Argumente an den aktuellen Kontext bindet und Platzhalter erklärt.

`function` kann einem Funktionsobjekt einen Namen zuweisen, sodass ein aufrufbares Objekt entsteht.

Grimms Kommentar zu diesem Thema in Abb. 2.

> Aufgabe:
>
> - Compilieren und Analysieren Sie folgendes Beispiel[12]: 20-29 (`bindAndFunction.cpp`)
> - Stellen Sie das Beispiel auf `auto`/Lambdafunktion um.

3.4 Move vs. Copy

Problem: C++ kopiert viel zu viel:

- Objekte werden in die Container kopiert

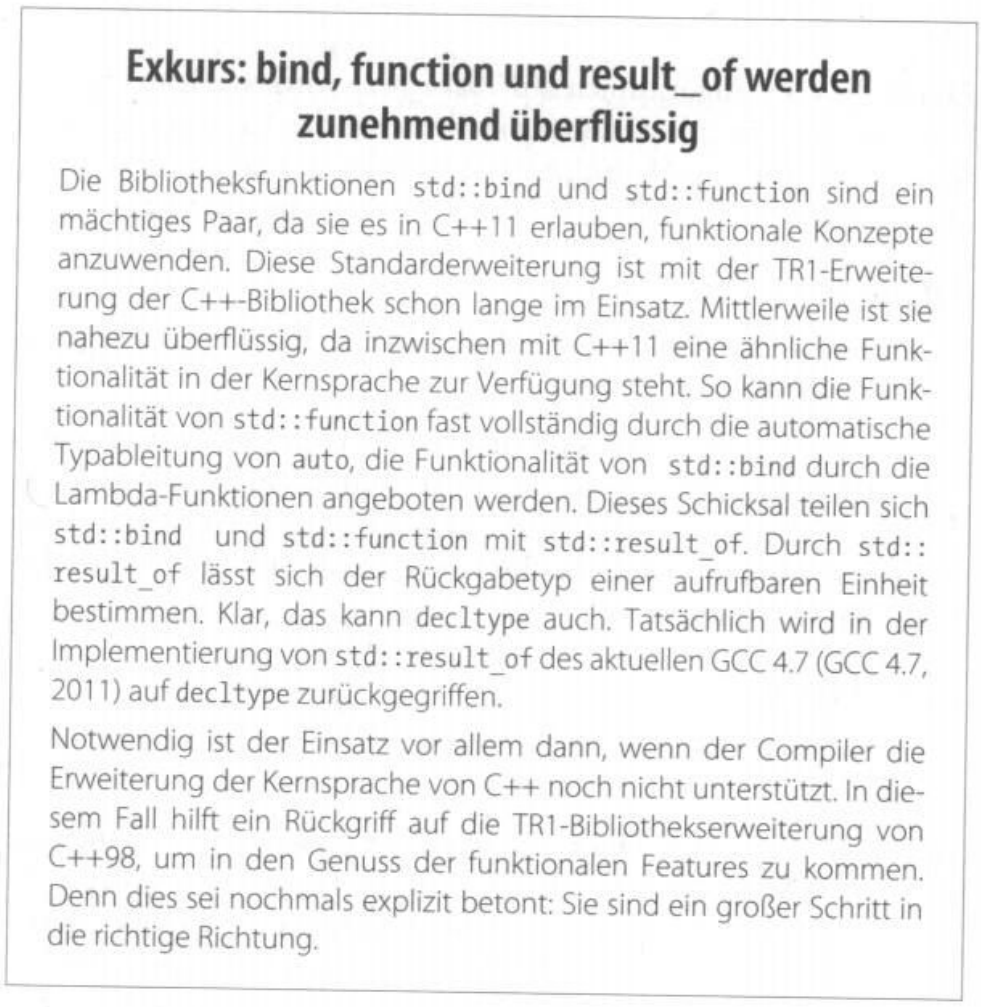
  Hat das Objekt nach dem Kopiervorgang noch eine Verwendung?

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

  Referenzen werden kopiert + Garbage-Collector

Move ermöglicht performanteres Übertragen von Objekten: Abb. 3.

[11] Grimm: S. 456
[12] Grimm
[13] Grimm: S. 32



Exkurs: bind, function und result_of werden zunehmend überflüssig

Die Bibliotheksfunktionen `std::bind` und `std::function` sind ein mächtiges Paar, da sie es in C++11 erlauben, funktionale Konzepte anzuwenden. Diese Standarderweiterung ist mit der TR1-Erweiterung der C++-Bibliothek schon lange im Einsatz. Mittlerweile ist sie nahezu überflüssig, da inzwischen mit C++11 eine ähnliche Funktionalität in der Kernsprache zur Verfügung steht. So kann die Funktionalität von `std::function` fast vollständig durch die automatische Typableitung von `auto`, die Funktionalität von `std::bind` durch die Lambda-Funktionen angeboten werden. Dieses Schicksal teilen sich `std::bind` und `std::function` mit `std::result_of`. Durch `std::result_of` lässt sich der Rückgabetyp einer aufrufbaren Einheit bestimmen. Klar, das kann `decltype` auch. Tatsächlich wird in der Implementierung von `std::result_of` des aktuellen GCC 4.7 (GCC 4.7, 2011) auf `decltype` zurückgegriffen.

Notwendig ist der Einsatz vor allem dann, wenn der Compiler die Erweiterung der Kernsprache von C++ noch nicht unterstützt. In diesem Fall hilft ein Rückgriff auf die TR1-Bibliothekserweiterung von C++98, um in den Genuss der funktionalen Features zu kommen. Denn dies sei nochmals explizit betont: Sie sind ein großer Schritt in die richtige Richtung.

Abbildung 2: bind und function[11]

- 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();`

Zusammen mit den *move*-Signaturen ergeben sich folgende fünf wichtige Standardsiganturen („die großen Fünf"):

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

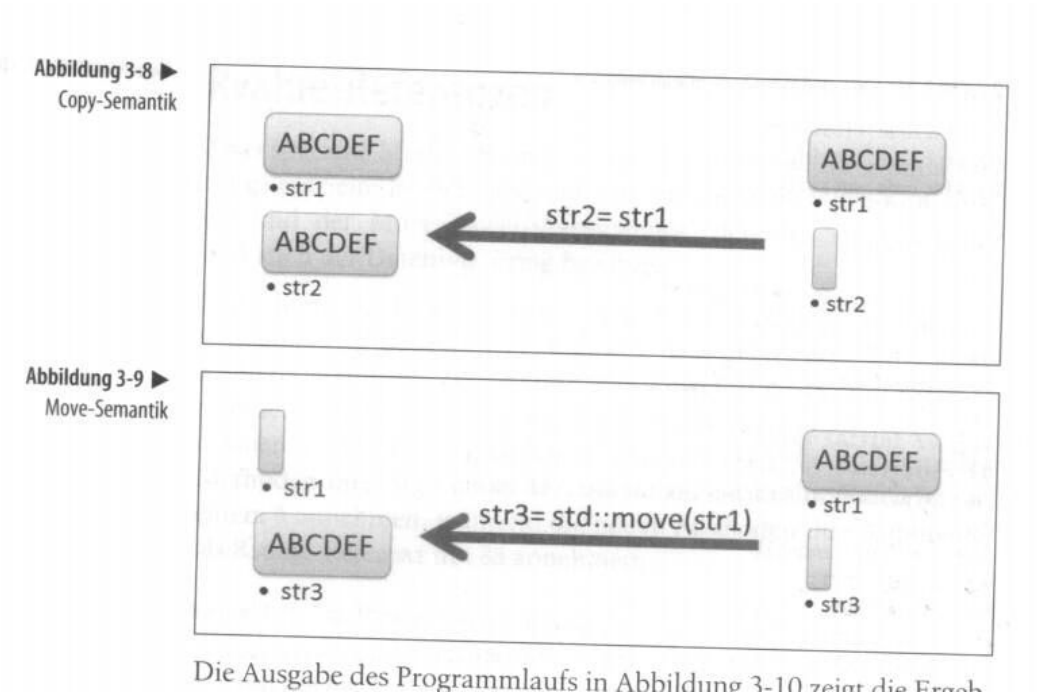


Abbildung 3: Move vs. Copy[13]

- Move-Zuweisung (C++11): `Class& operator=(Class&&)`

Zu beachten sind dabei folgende Regeln:

- *Regel der großen Fünf*: Wenn einer der großen Fünf implementiert wird, müssen alle fünf implementiert werden.
- *Nullregel*: Es ist besser keinen der großen Fünf zu implementieren[14].

Sollen einzelne der implizit durch den Compiler generierten Operationen verboten werden, kann dies mit `delete` erfolgen.

```
class Simple{
private:
  int *daten{nullptr}; // Roher Zeiger !!!
  Simple( int n ) : daten{ new int[n] } {}
  ~Simple() { delete[] daten; }
  Simple(const Simple&) = delete;  // keine Kopie
  Simple& operator=(const Simple&)=delete; // keine Zuweisung
  Simple(Simple&&) = delete;  // kein Verschieben
  Simple& operator=(Simple&&)=delete; // keine Verschiebzuweisung
};
```

Aufgabe:

Compilieren und Analysieren Sie `Move.cpp`. 5x wird ein großes Array kopiert. Das soll im Folgenden verhindert werden:

- Ergänzen Sie in der Klassendefinition den Move-Konstruktor und die Move-Zuweisung.

[14] Wolf: 291.

Jetzt sollten es nur noch zwei/drei (je nach Compiler) Kopiervorgänge sein.

- An einer Stelle können Sie durch Einfügen eines `std::move` ein weiteres Kopieren verhindern.

Die Möglichkeit Freispeicher durch eine andere Instanz übernehmen zu lassen, verhindert zwar oftmals unnötiges Kopieren. Schöner wäre es aber, Instanzen direkt im Speicher des Containers instanzieren zu können. Die neuen `emplace`-Funktionen der Standard-Container ermöglichen dies. Bei einem `emplace` werden die Parameter so angegeben, wie sie für einen Konstruktoraufruf benötigt werden: Beispiel für `vector`[15]

Aufgabe:

Ändern Sie das *Move*-Beispiel so ab, dass durch `emplace_back` ein *move* in den `vector` vermieden wird.

3.5 Verschiedenes

Definition von Typen

Ein weiteres neues Sprachelement ist `using` als Ersatz von `typedef`:

```
typedef struct my_struct my_type;
// ist gleichbedeutend mit
using my_type = struct my_struct;
```

Mit `using` lassen sich Templates (teilweise) binden[16]:

```
template <typename T, int Line, int Column>
class Matrix;

template <typename T, int Line>
using Square = Matrix<T, Line, Line>;

template <typename T, int Line>
using Vector = Matrix<T, Line, 1>;
```

Typsichere Aufzählungen

`enum class`:

- Typsicher, da nicht implizit zu `int` konvertierbar
- Keine *namespace pollution*
- ABER: Keine Rückkonvertierung zum Namen möglich

[15] http://www.cplusplus.com/reference/vector/vector/emplace/
[16] Grimm: S. 186

[Kernsprache/enum.cpp][17]

Neue Literale

- Raw Strings[18]
  `R"Trenner(raw string)Trenner"`
  `Trenner`: beliebige (auch leere) Zeichenkette

- Benutzerdefinierte Literale
  Literale der Form: `wert_einheit` (`25.6_km, 2345_s, 2.2e24_kg`)
  Beispiel: [Kernsprache/Aufgaben/userDefinedLiterale.cpp][19]
- Binäre Literale: Mit C++14 lassen sich Literale der Form `0b001010011` schreiben.
- Digit Seperator: Zur Lesbarkeit ist es möglich das einfache Hochkomma einzufügen:
  `0b0'0101'0011, 123'345'678`

Automatischer Rückgabetyp

Bei der Deklaration/Definition von Funktionen kann auch das Schlüsselwort `auto` für den Typ verwendet werden. In diesem Fall bestimmt das erste `return`-Statement den Rückgabetyp.[20]

```
auto Correct(int i) {
  if (i == 1)
    return i;               // return type deduced as int
  else
    return Correct(i-1)+i;  // ok to call it now
}
```

3.6 Variadic Templates

Templates mit variabler Zahl von Parametern

- Element zur funktionalen Programmierung

  ```
  template <typename ... Args>
  function (Args ... args)
  ```

- `args` ist dabei ein „Parameter Pack"
- `args...` ist die entpackte Parameterliste

[17] Grimm: S. 202f.
[18] Grimm: S. 205f.
[19] Grimm: S. 213f.
[20] https://en.wikipedia.org/wiki/C%2B%2B14#Function_return_type_deduction (24.8.2018)

[wagner/variadic.cpp]

Aufgabe:

Schreiben Sie eine Funktion `bool split(istream,char,...)`, die von einem komma-separieten *input stream* Werte liest und in die gegebenen Typen der Parameterleiste umwandelt und zurück gibt. Testen Sie den Code mit einer Zeile aus `medien.csv`:
`split(istream,char,string&,string&,string&,char&,int&,int&)`

Die Funktion soll dann folgendermaßen aufgerufen werden:

```
string signatur, autor, titel;
char typ;
int seitenzahl, spieldauer;
stringstream ss ("A01,Heinz Autor,Der Titel,B,125,0");
bool ok = split(ss,',',signatur,autor,titel,typ,seitenzahl,spieldauer);
```

Verwenden Sie in `MedienverwaltungClass::load` die neue Funktion statt dem `tokenize`.

4 Neues in der Standardbibliothek

4.1 Container-Templates

Tupel

`tuple` ist die Verallgemeinerung von `pair`. Es nimmt mehrere Daten verschienenen Typs auf.

Für `tuple` existieren diverse Hilfsfunktionen in der Standardbibliothek:
.[wagner/tuple.cpp]

- Z. 10: `make_tuple` erzeugt ein Tupel aus einzelnen Werten.
- Z. 23: Zusammen mit dem Referenzwrapper `ref/cref` wird ein Tupel aus Referenzen erzeugt.
- Z. 50: `tie` erzeugt ein Tupel aus Referenzen für alle Werte. Dies ist eine einfache Möglichkeit die Werte eines Tupels Variablen zuzuweisen.
- Z. 79: Deklaration von Tupelvariablen + Zuweisung

Eine geschickte Anwendungsmöglichkeit für das Tupel ist die Rückgabe mehrerer Werte aus einer Funktion:

```
tuple<int,double,string,bool> returnFourValues() {
  int a = 5;
  double b = 10.2;
  std::string c = "test";
  bool d = true;
  return make_tuple(a,b,c,d) ;
}
```

Der klassische C++-Weg bestand darin, eine Struktur zu definieren, die die vier Typen bindet, und diese Struktur als Rückgabewert zu verwenden.

Referencewrapper

Der Referenz-Wrapper ist auch bei der Lösung des klassischen Problems hilfreich, dass Container keine Referenzen enthalten können. Der Container kann den Referenz-Wrapper als Element haben, dieser muss aber mit `emplace` direkt im Container erzeugt und belegt werden. Der Zugriff auf die Referenz selbst erfolt mit der Methode `get()` des `reference_wrapper`. [`TourDeC++11/referenceWrapper.cpp`]

> Aufgabe:
>
> Die Datenspeicherung in `MedienverwaltungClass` hat den Nachteil, dass dort rohe Zeiger abgelegt sind. Rohe Zeiger gelten im modernen C++ als „böse". Die C++-Container erlauben aber keine Ablage von Vererbungshierarchien. Eine Möglichkeit dieses Problem zu lösen wäre folgender Ansatz:
>
> Ändern Sie `MedienverwaltungClass` so ab, dass sie die Daten in drei Containern speichert:
>
> - `vector<Buch>` mit allen Büchern
> - `vector<CD>` mit allen CDs
> - `map<string,reference_wrapper<MediumBase> >` mit Referenzen auf alle Bücher und CDs
>
> Passen Sie die Klasse entsprechend an.

Array

> `std::array` lässt sich kurz und knapp charakterisieren: `std::array` vereint die Speicher- und Laufzeitanforderungen des C-Arrays mit einem STL-konformen Interface. Das Beste aus beiden Welten.[21]

Für `array` existieren diverse Hilfsfunktionen in der Standardbibliothek:
.[`DieStandardbibliothek/arrayInterface.cpp`][22]

- Z. 17, 42: Ausgabe über `std::copy`
- Z. 33: Ausgabe über Range-For
- Z. 31: `{{...}}`: geht auch mit einfachen Klammern, war bei ersten Compilern so nötig, um den Aufruf des Konstruktors zu umgehen.
- Z. 47: `std::accumulate`, was in Z. 48 zur Bildung des Mittelwertes genutzt wird.
- Z. 58: `std::swap` vertauscht Arrays.
- Z. 70: Zugriff wie auf `tuple` möglich

[21] Grimm: S. 424.
[22] Grimm: S. 424.

Die Headerdateien der angegeben Funktionen finden sich in `<algorithm>` und `<numeric>`.

Aufgabe:

`DieStandardbibliothek/arrayInterface.cpp`:
Greifen Sie über die Indexgrenzen des Arrays hinaus. Greifen Sie einmal mit dem Indexoperator `[]` bzw. mit der `at()`-Elementfunktion zu.

Einfach verkettete Liste

> `std::forward_list` ist ein sequenzieller Container mit einem eingeschränkten Interface. Als einfach verkettete Liste benötigt sie nicht mehr Speicher als die entsprechende C-Datenstruktur. `std::forward_list` ist für den speziellen Einsatz konzipiert: Wenn die optimierte Speicheranforderung, das schnelle Einfügen oder Entfernen von Elementen gefragt ist und der wahlfreie Zugriff nicht benötigt wird, ...[23]

Für `forward_list` existieren diverse Memberfunktionen. Im Beispiel `forwardListManipulate.cpp`[24] wird die Verwendung vorgestellt:

- Z. 12: `.push_front()` stellt ein neues Element an den Anfang der Liste.
- Z. 27: `.erase_after()` löscht das Element nach der Iteratorposition.
- Z. 31: `.insert_after()` fügt ein Element nach der Iteratorposition ein.
- Z. 40: `.splice_after()` fügt eine Liste in eine andere ein.
- Z. 46: `.sort()` sortiert die Liste.
- Z. 52: `.reverse` kehrt die Reihenfolge um.
- Z. 58: `.unique()` entfernt direkt aufeinanderfolgende Duplikate.

Aufgabe[25]:

- Bestimmen Sie die Anzahl der Elemente einer `forward_list`.
- Betrachten Sie die Musterlösung `forwardListSize.cpp`: Die dort implementierte Templatefunktion `sizeOf` funktioniert prinzipiell auch für `vector`.
  - Verallgemeinern Sie die Templatefunktion so, dass es auch für einen `vector` funktioniert.
  - Ergänzen Sie eine Spezialisierung des Templates für `vector`, die die Größe des Vectors aus der Differenz der Iteratoren (Anfang/Ende) berechnet.
- Bestimmen Sie die Größe einer `forward_list` unter Verwendung von `count_if`.

[23] Grimm: S. 428
[24] Grimm: Beispiel 20-22, S. 429.
[25] Grimm: Aufg. 20-11 S. 431.

Hashtabellen

Mit C++98 standen mit `set`, `multiset`, `map` und `multimap` Container zur Verfügung, die sich wie Hashtabellen anfühlen, aber sortierte Schlüssel haben. Die Zugriffszeiten sind daher schlechter als bei Hashtabellen. Mit C++11 gibt es nun die vier Container mit dem `unordered_`-Prefix. Diese Implementierungen sind Hashtabellen.

Aufgabe[26]:

Vergleichen Sie die Zugriffszeit von `map` und `unordered_map`.

Legen Sie zwei große assoziative Container vom Datentyp `map<int,int>` und `unordered_map<int,int>` an und belegen sie diese mit Werten.

Der zufällige Zugriff auf die Elemente erfolt nach Einführung der Zufallszahlen, die Zeitmessung wird bis zur Einführung der Zeitbibliothek zurückgestellt.

4.2 Algorithmen

Neue Algorithmen

Neue Algorithmen[27] → `all_of, any_of, copy_if, is_sorted, minmax_element`

Mit C++17 soll begonnen werden die Parallelisieung der Algorithmen zu standardisieren.

Aufgabe:

Erweitern Sie die Büchereiverwaltung um eine Funktion „Alle CDs anzeigen":

- Ergänzen Sie `MedienverwaltungClass` um eine Methode `map<string,CD> getCDs()`, die unter Verwendung von `std::copy_if` die die CDs aus der Gesamtmap kopiert.
- Passen Sie `Bucherei.cpp` und `MedienUI.cpp` entsprechend an.

4.3 Neue Funktionalität

Smart Pointer

- `auto_ptr`: C++03 Implementierung: Keine Unterscheidung von move und copy → passt nicht ins C++11-Konzept

[26] Grimm: Aufg. 20-13 S. 447: `mapHashComparison.cpp`.
[27] http://www.cplusplus.com/reference/algorithm/

- `unique_ptr` löst `auto_ptr` ab.
  .[`TourDeC++11/uniquePtrMove.cpp`]
- `shared_ptr`: arbeitet mit Referenzzähler
  .[`anhang_a/smart_ptr01.cpp`] (Beispiel mit `Deleter`)
- `weak_ptr`: Wrapper um `shared_ptr`: „Ein nicht aktivierter `shared_ptr`". Die „Aktivierung" erfolgt durch `.lock()`. Kann für den Umgang mit zyklischen Referenzen nützlich sein (s. Abbildung 4)
  .[`TourDeC++11/sharedWeakPtr.cpp`, `wagner/ZyklischeReferenz.cpp`][28]

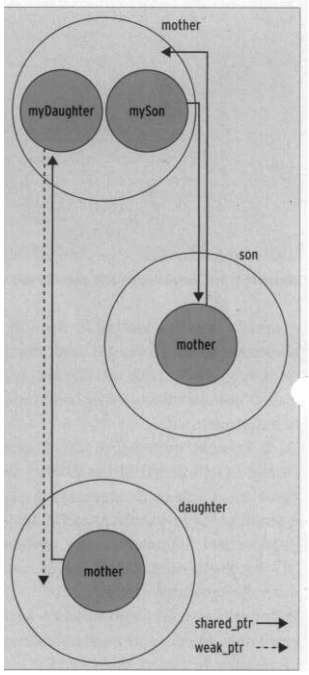


Abbildung 4: Zyklische Referenz[29]

Aufgabe:

Ändern Sie die Implementierung von `createMedium` und die Verwendung in `Bucherei.cpp` so ab, dass über die Verwendung von `unique_ptr` keine rohen Zeiger mehr auftreten.

Reguläre Ausdrücke

Prinzipielles Vorgehen

- Erkläre den regulären Ausdruck
  `std::regex r(<muster>);`
- Prüfe, ob die Zeichenkette exakt dem Muster entspricht:
  `std::regex_match (<str>,r);`

[28] Listing 4 aus LM 03/2013
[29] Linux Magazin 04/2013, S. 106.

- Halte das Ergebnis der Suche
  `std::smatch s;`
  `std::regex_search(<str>, s, r);`: `s` enthält nun die Treffer.
  .[`anhang_a/regexpr.cpp`]
- Suche und ersetze Treffer:
  `std::regex_replace (<str>,r,<repl>);`
  Über `$1`, `$2`, `...` kann dabei auf Gruppen des regulären Ausdrucks zugegriffen werden.

Aufgabe:

Schreiben Sie eine Funktion „Datenimport", die

- eine Datei im Format von `wagner/import.txt` liest,
- darin die nötigen Informationen unter Verwendung von regulären Ausdrücken extrahiert,
- für jedes Medium eine freie Signatur nach dem Muster
  `<erster Buchstabe Autor><ersterBuchstabe Titel><laufende Nummer>` sucht,
- das Medium dem Datenbestand hinzufügt.

Anmerkung: Es gibt einen REGEX-Tester[30]

Zufallszahlen

Die Erzeugung von Zufallszahlen erfolgt nach folgendem Muster:

- Erzeugung eines Zufallszahlenstroms
- Abbildung auf die gewünschte Verteilungsfunktion
- Abrufen der Zufallszahlen

Im Beispiel `[DieStandardbibliothek/distribution.cpp]` wird die Verwendung vorgestellt:

- Z. 25/28: Definition des Zufallszahlenstroms
- Z. 33-39: Abbildung auf verschiedene Verteilungsfunktionen
- Z. 47-50: Abrufen der Zufallszahlen

Aufgabe[31]:

Ergänzen Sie die `map/unordered_map`-Übung:

- Greifen Sie in einer Schleife auf `mapSize/10` zufällige Schlüssel zu und geben Sie diese aus.

[30] http://regex101.com
[31] Teil von `DieStandardbibliothek/Aufgaben/mapHashComparison.cpp`

Zeitbibliothek

Die C++11-Zeitbibliothek kennt folgende wichtigen Klassen:

- Zeitpunkt `std::chrono::time_point`
- Dauer `std::chrono::duration`
- Zeitgeber `std::chrono::system_clock`

Mit folgendem Code lässt sich daher eine Zeitmessung bauen:

```
auto start = std::chrono::system_clock::now();
// code for measurement
std::chrono::duration<double> dur=std::chrono::system_clock::now()-start;
std::cout << "time: " << dur.count() << " seconds" << std::endl;
```

> Aufgabe:
>
> Messen Sie das Zeitverhalten von Maps mit sortierten Schlüsseln und Hashmaps.

Typvarianten[32]

Alternative Typen konnten bisher über `union` abgebildet werden. Dabei blieb es aber dem Entwickler überlassen, wie er sicher stellen will, welcher Typ gerade aktuell ist. `std::variant` gibt hier Unterstützung:

```
using ItsOk = std::variant<std::vector<int>, double>;

int main() {
  //set the variant to vector, this constructs the internal vector
  ItsOk io = std::vector{22, 44, 66};
  // reset to double - the internal vector is properly destroyed
  io = 13.7;
  // There's no vector in the variant - throws an exception
  int i = std::get<std::vector<int>>(io)[2];
}
```

Die Verwendung von gemeinsamen Eigenschaften aller Typen erfolgt über `std::visit(<function>, <variant>)`:

```
std::ostream& operator<< (std::ostream& os, MyVariant const& v) {
  // all variants have to implement the stream operator
  std::visit([&os](auto const& e){ os << e; }, v);
  return os;
}
```

Welcher Typ gerade „geladen" ist, kann über die Funktion `.index()` abgefragt werden.

Typvarianten und Templates ermöglichen ein Programmierparadigma, das *duck typing* genannt wird. Es werden Funktionen für Typen geschrieben, die bestimmte gemeinsame Eigenschaften aufweisen müssen, die nicht formal über eine Vererbungshierarchie gesichert werden.

[32] https://arne-mertz.de/2018/05/modern-c-features-stdvariant-and-stdvisit/ (6.1.2019)

Aufgabe:

- Prüfen Sie, ob das Beispiel `wagner/variant.cpp` bei Ihnen lauffähig ist.
- Wandeln Sie die Funktion „Alle Medien anzeigen" in Richtung Typvarianten (statt Vererbung) um. Die Funktion `createMedium()` soll dabei einen `variant<string,variant<Buch,CD> >` zurückgeben.

Optionale Werte[33]

C++17 führt den Template-Typ `std::optional<T>` ein. Mit diesem Konstrukt lässt sich das Problem optionaler Werte, wie sie beispielsweise aus Datenbanken (`NULL`-Werte) oder Konfigurationen zurückgegeben werden. Bisher konnte ein nicht-vorhandener Wert entweder über „magische" Werte (`-255` bedeutet „nicht vorhanden") oder über Zeiger (`nullptr` bedeutet „nicht vorhanden") abgebildet werden.

Eigenschaften von `std::optional<T>`:

- Erzeugen eines optionalen Wertes: `std::make_optional<T>()`
- `NULL`-Wert: `std::nullopt`
- Existenzprüfung: `has_value()`
- Wert auslesen: `value()`. Wird `value()` aufgerufen, wenn gar kein Wert vorhanden ist, so wird `bad_optional_access` geworfen.
- Abfrage mit Defaultwert: `value_or(T&& default)`
- Rücksetzen eines Wertes: `reset()`
- `optional` funktioniert nur mit einfachen Klassen. Eine Verwendung in einer Vererbungshierarchie ist nicht möglich (z.B. `MedienverwaltungClass::getMedium()`).

Aufgabe:

- Prüfen Sie, ob das Beispiel `wagner/optional_variant.cpp` bei Ihnen lauffähig ist.
- Fügen Sie nun auch die Funktion „Medium anzeigen" hinzu. `getMedium()` soll einen `optional<variant<Buch,CD> >` zurückliefern.

[33] https://arne-mertz.de/2018/06/modern-c-features-stdoptional/ (6.1.2019)

Unbestimmter Typ

Der Typ `any` ist die typsichere Variante des `void*`. Programmschichten, die Daten nur durchreichen, ist es irrelevant welcher Typ genau weitergereicht wird. Mit dem Typ `any` können Transportdaten einfach durch Zuweisung eingepackt werden und mit `any_cast` in den Ursprungstyp wieder rückgewandelt werden.

Beispiel[34]

[`wagner/any.cpp`]

Typinformationen

Da mit den Templates, `auto` und `decltype` die starre Typisierung immer mehr aufgebrochen wird, existieren mit C++11 erweiterte Möglichkeiten Eigenschaften von Typen zur Compilezeit zu überprüfen.

Alle C++-Typen gehören exakt einer der folgenden Typenkategorien an:[35]

```
is_void
is_integral
is_floating_point
is_array
is_pointer
is_reference
is_member_object_pointer
is_member_function_pointer
is_enum
is_union
is_class
is_function
```

Davon abgeleitet gibt es zusammengesetzte Kategorien. So bedeutet `is_arithmethic`: `is_integral ODER is_floating_point`.[36] [`DieStandardbibliothek/typeCategories.cpp`]

Desweiteren können Typeigenschaften abgefragt werden: `is_abstract, is_polymorphic, is_signed, ...`

Überblick[37]

> Aufgabe:
>
> Ändern Sie die Implementierung des Templates `015/Templates002/main.cpp` (`bigNum`) so ab, dass
>
> - im Falle von numerischen Argumenten der Kleinervergleich angewendet wird.
> - in allen anderen Fällen das Ergebnis von `.size()` verglichen wird.
>
> Für die Lösung gibt es zwei Ansätze:

[34] https://en.cppreference.com/w/cpp/utility/any
[35] Grimm: S. 343.
[36] Grimm: S. 345.
[37] http://www.cplusplus.com/reference/type_traits/|

- `is_arithmethic<T>::type` liefert `true_type` oder `false_type`. Damit lässt sich ein weites Funktionstemplate aufrufen, das als 3. Parameter `true_type` oder `false_type` nimmt.
- `is_arithmethic<T>::value` liefert `true` oder `false`. Damit lässt sich mit `if constexpr` innerhalb des Templates verzweigen.

Konzepte (C++20)

Die Verwendung von Typvarianten hat einen Nachteil: Es ist nirgens definiert, welche die gemeinsamen Typeigenschaften sind, die beispielsweise für ein `visit` vorausgesetzt werden. Diese Lücke schließt die Verwendung von Konzepten. Hier kann ich definieren, welche Eigenschaften von einem Typ in einem bestimmten Kontext verlangt werden. In C++ lässt sich dann dieses Konzept wie ein Typ verwenden. Unter der Haube wird ein Template definiert, das bei einer Instanzierung die entsprechenden Typeigenschaften prüft.

[wagner/concept.cpp]

5 Multithreading

Der Lebenszyklus eines Threads lässt sich wie folgt kontrollieren:

- Erzeugen eines Threads:
  `std::thread t (<funktion>, <params>, ...);`

  Die Parameter werden grundsätzlich in den Thread kopiert. Sollen sie als Referenz übergeben werden, muss dies durch `std::ref` gekennzeichnet werden.
- Synchronisieren von Threads

  `t.join();`
- Entkoppeln von Threads

  `t.detach();`: Dies funktioniert zwischen Kind und Enkel-Thread. Der Hauptthread muss immer auf alle abgeleiteten Threads warten.

[anhang_a/thread01.cpp]

Die Sicherung von Ressourcen erfolgt über *Mutex* (*mutual exclusion*, wechselseitiger Ausschluss):

- `std::mutex m;` definiert Mutex
- `std::lock_guard<std::mutex> g(m);` setzt Mutex für die Lebenszeit von g oder bis g.unlock()

[anhang_a/thread02.cpp]

Ist hingegen eine Operation atomar, so ist eine Sicherung nicht nötig. Eine atomare Operation ist eine Operation auf einen atomaren Datentypen. Als atomare Datentypen gibt es einerseits die *built in* Typen `atomic_bool, atomic_char, ...`, andererseits können auch benutzerdefinierte Typen als atomar gekennzeichnet werden, indem sie in das Template `atomic<T>` genommen werden.

atomic<T>[38]

Threads lassen sich über Semaphore synchronisieren. Der „Sender" signalisiert dem „Arbeiter", wann er weitermachen darf. Bestandteile:

- Einfache (boolsche) Variable: Das Semaphor
- `std::lock_guard` zum Setzen des Semaphors
- `std::unique_guard` zum Lesen des Semaphors
- `std::condition_variable` zum Triggern des Arbeiters durch den Sender

> Aufgabe:
>
> Erweitern Sie das Beispiel `TourDeC++11/conditionVariable.cpp` auf mehrere *worker*.

Threads lassen sich auch über `future/promise`, d.h. über einen Rückgabewert synchronisieren:

```
    // promise definieren
             promize prom;
       // future ableiten
fut = prom.get_future();
        // thread starten
  thread t(f, prom, ...)  | f(prom, ...)
                          | // thread-Bearbeitung
                          | // Ergebnis-R"uckgabe
    // Ergebnis erwarten  | prom.set_value(e)
            e = fut.get()
```

[Multithreading/futurePromise.cpp]

Sollen mehrere Threads auf dasselbe Ergebnis warten, kann dies mit einem *shared future* erfolgen.

[sharedFuture.cpp]

Der Future/Promise-Mechanismus lässt sich über die `async`-Funktion vereinfachen:

- `auto f = std::async (launch::async,<funktion>, <params>);` liefert ein Future-Objekt
- Synchronisation über `f.get()`

[38] http://www.cplusplus.com/reference/atomic/atomic/

Je nach Programmierstil enthalten Programme globale Variablen. Diese verhindern, dass ein Programm in mehreren Threads laufen kann. Eine Möglichkeit ein solches Programm *threading*-fähig zu machen, ist, Variablen, die nun pro Thread benötigt werden, als „threadlokal" zu definieren. Threadlokale Daten bilden ein Array, in welchem jeder Thread sein eigenes Element hat: [`Multithreading/threadLocal.cpp`][39]

Aufgabe:

- Wandeln Sie das Beispiel `Multithreading/futurePromise.cpp` in einen `std::async`-Aufruf um.
- Parallelisieren Sie das Beispiel `wagner/dotProduct.cpp`[40], indem Sie die Berechnung auf vier *threads* verteilen.

Coroutinen (C++20)

Mit C++20 kommen auch Coroutinen. Dafür existiert aber noch keine standardkonforme Compilerimplementierung. Hier aber ein Beispiel, das erahnen lässt, in welche Richtung die Entwicklung läuft:[41]

```
// will lazily generate numbers from 0 to 9
cppcoro::generator<std::size_t> getTenNumbers()
{
    std::size_t n{0};
    while (n != 10)
    {
        co_yield n++;
    }
}

void printNumbers()
{
    for(const auto n : getTenNumbers())
    {
        std::cout << n;
    }
}
```

6 Quellen

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

[39] Grimm: S. 269
[40] Linux Magazin 04/2012
[41] https://oleksandrkvl.github.io/2021/04/02/cpp-20-overview.html#coroutines (8.4.2022)